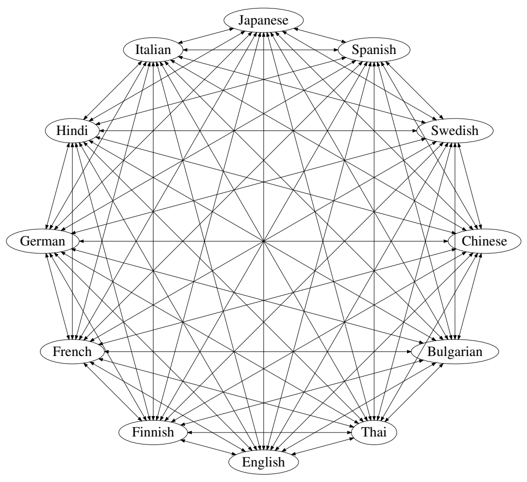
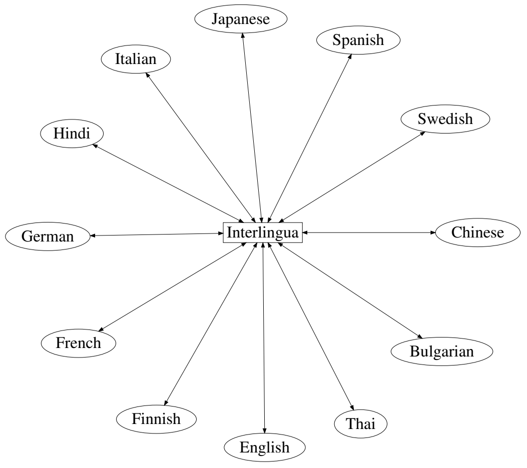
Computational Grammar
An Interlingual Perspective





Aarne Ranta

March 22, 2024

Contents

Preface

This book is an introduction to the concepts of grammar and their implementation on a computer. It is a free-standing sequel to *Grammatical Framework: Programming with Multilingual Grammars* (CSLI 2011, “the GF book”). However, in the decade that has passed since finishing the GF book, GF has expanded from special-purpose controlled languages to general-purpose processing of natural language, which is not covered in the GF book. At the same time, education on grammar has become less wide-spread in schools and universities. To fill the gap, a text of book size that explains the concepts from the beginning, is more accessible than scattered introductions in research papers. Our ambition is that it could serve as a general introduction to grammar to computationally or mathematically minded readers, even if their goal is not to write grammars themselves.

Since GF details can be gathered from the GF book and from free online documentation, the focus in this book is on how grammar is used for describing natural language rather than on the details of the formalism. After years of experience with dozens of languages, we exploit the interlingual perspective of GF and developed it further, and also put it into comparison with other approaches. The most important of these is the Universal Dependencies (UD) programme.

UD also shares with GF the idea of common descriptions for multiple languages. UD has also shown how grammatical concepts fit into data-driven language processing. We will make use of UD concepts parallel to GF as a bridge to data-driven approaches. In a nutshell, we have found UD useful when analysing language, whereas the additional expressivity of GF is useful for language generation, translation, and semantics.

Gothenburg 15 March 2020 (first version); March 22, 2024(current version)

Aarne Ranta

Reading guide

The first four chapters are a non-technical introduction to computational grammars, not presupposing any GF at all. In particular, Chapter 3 can be used as an introduction to UD and enable readers to contribute to UD treebanks. It is at the time of writing the only textbook-like introduction to Universal Dependencies.

The GF part starts in Chapter 5. The GF formalism is introduced in a hands-on fashion, meant to be sufficient for reading all the code in the book. A reader used to programming by example rather than by tutorials or manuals should be able also to catch up the knowledge needed for her own GF projects. But the GF book (or the on-line tutorial and reference manual) are recommended for readers who want a solid understanding of programming in GF.

Chapters 5 and 6 discuss the general principles of morphology, lexicon, and syntax implementation in GF. Chapter 7 takes a closer look at different languages, focusing on more or less well-known peculiarities of them. Linguistically oriented readers who are not aiming to program themselves should be able to read this chapter to get more insight into how seemingly huge differences of languages can be understood within a shared interlingual structure.

Chapters 8 and 9 are, just like Chapter 7, independent of the coding details of morphology and syntax. They are aimed to put grammar into a wider perspective and show some typical uses of it, all tested in real-world applications after the publication of the GF book.

Chapter 10 completes the presentation by looking at the algorithms underlying grammar-based language processing. Some of it is well-known material from language theory books, but it is here intimately connected to the previous material.

All of the chapters can naturally be read in sequence. But there are subsets that can be relevant for readers with more particular interests:

- general interest in grammar and languages: Chapters 1–4, 7
- writing grammars for new languages: Chapters 1–7
- building grammar-based systems: Chapters 1–4, 8, 9

Chapter 1

Introduction

In this chapter, we will first discuss the scope of grammar, as well as some arguments for and against using grammar in natural language processing. We will discuss questions such as normative vs. descriptive and theoretical vs. empirical. The main conclusion is that grammar is useful, but some common ways of understanding it are not fruitful (Section 1.1).

Section 1.2 is a miniature of much of the rest of the book, walking through a simple example and discussing its grammatical analysis on different levels: tokens, morphology, syntax. Many of the basic concepts are introduced here, but they will be repeated in later chapters.

Section 1.3 introduces the idea that different languages can share a grammatical structure. This idea is both intellectually fascinating and useful in practice. Even though most of the examples in the book are from English, we will continuously mention other languages to cover the issues that might pose problems to the interlingual approach.

1.1 Why grammar

A grammar in the usual sense is a set of rules describing a language. One common view of grammar is that it is **normative**: the rules state what one may and may not say or write. A normative grammar of English might, for instance, forbid the use of "split infinitives":

> Don't say *You must learn to not split infinitives*,
> say *You must learn not to split infinitives.*

Normative grammars are in contrast to **descriptive** grammars, which describe how language is actually used. A descriptive grammar rule might say, for instance,

> There are two ways to form the negation of an infinitive: *to*+*not*+verb (the split infinitive), and *not*+*to*+verb (the full infinitive)

Recognizing both kinds of infinitives is necessary when one wants to analyse actual language, where both split and full infinitives do occur. In the science of linguistics, theoretical as well as computational, descriptive grammar is the completely dominating approach to grammar.

Nonetheless, encoding the normative values of different constructions is an important part of descriptive linguistics. Thus one might want to mark split and full infinitives in special ways and say that certain kinds of style avoid the one or the other. In **Natural Language Processing** (NLP), such norms are of particular importance when **generating** language. For instance, when translating from one language to another, one should maintain the style of the original. An extreme example is if some word or construction is considered "vulgar": then the translation should definitely not use it when rendering "standard" language. On the other hand, when a vulgar expression appears in a novel or in a film, then the natural way to translate it is with a corresponding vulgar expression.

In this book, we will focus on descriptive grammar, but keep in mind the need of describing and enforcing language norms — as long as they are a part of a complete language description.

Assuming that the goal is to write a descriptive grammar, the next question is its **completeness**. The ambition in much of modern linguistics is to write grammars that *exactly* match the language — that allow "all and only" the "grammatically correct" expressions; "grammatically correct" is of course not taken in a normative sense, but in the sense "acceptable by competitive language users". Hence a grammar is a theory that can be tested by comparing it to observed language use by competitive speakers and writers.

The completeness property is formalized in the mathematical theory of **formal languages**. In this theory,

- a **language** L is a set of sentences
- a **sentence** of L is an element of the set L
- a **grammar** $\mathrm{G}(L)$ is a system of rules that produces all and only the sentences of L

The simplest possible grammar is a list of all sentence. For example, traffic lights can be seen as a language that has three sentences: *red*, *yellow*, and *green*, plus perhaps three more *red-yellow*, *yellow-green* and *blinking yellow*. However, in the general case, a language is an *infinite* set of sentences. For example, the set of expressions for natural numbers: 0,1,2,3,...,78409,... is infinite, as there is no largest number. Therefore, one cannot list all the sentences, but the grammar must be a set of **production rules**, which **generate** more and more numbers. A simple "grammar" of number expressions would have two rules:

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are numbers
- if X and Y are numbers, then XY is a number

According to this grammar, any sequence of digits is a number. However, this rule would be **overgenerating**, as it would also generate sequences such as 012, which we don't want to count as number expressions. A more accurate grammar is the following:

- 0 is a number
- 1, 2, 3, 4, 5, 6, 7, 8, 9 are positive numbers
- positive numbers are numbers
- if P is a positive number and D is 0 or a positive number, then PD is a positive number

Natural languages are infinite sets of sentences, in the sense that one can always add new words to a given sentence without loss of grammaticality. This book will therefore show how to write rules that cover infinite sets of natural language expressions. However, unlike linguists that follow the formal language paradigm, we will *not* attempt to cover "all and only" the grammatical sentences. The fundamental problem in that approach is that there is neither any mathematical definition, nor even a native speaker agreement, stating what exactly is a grammatically correct sentence in a natural language. The limits of grammaticality are vague, or, more accurately, **gradient**: some expressions are "more grammatical" than others.

A typical case is the length of a sentence: when more words are added, a sentence may become more and more difficult to understand, and thereby less and less easy to recognize as a valid sentence of the language. It has for intance been pointed out that **center embedding**, although "in principle" unlimited, never occurs at depth greater than 3:

The rat ate the malt.

The rat the cat killed ate the malt.

The rat the cat the dog chased killed ate the malt.

(Karlsson 2007) A possible reaction to this is to say, in the descriptive spirit, that English has no unrestricted center embedding, but only down to level 3. This would lead to a more complicated grammar than unrestricted embedding, because we would need three specific rules instead of one general rule; you can get an idea of this by writing a set of rules for number expressions that limits their size to for example six digits. Another reaction is to keep the general rule but — not unlike when describing language norms — encode the fact that its repeated use becomes less and less acceptable. This could then guide for instance a language generation system to select a better expression for the additional fact that the cow tossed the dog:

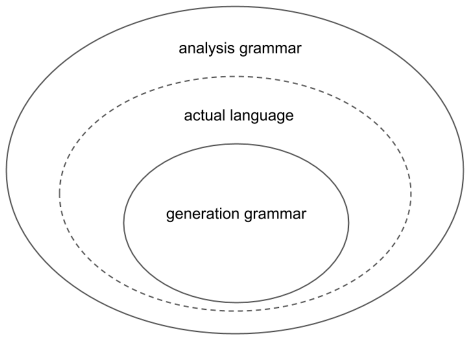
The malt was eaten by the rat that was killed by the cat that was chased by the dog that was tossed by the cow.

using **right embedding**, which permits more repetition than than medial embedding.

As a general strategy, we will draw a distinction between grammars that analyse and generate language:

- The **analysis** of language should be more permissive than "only grammatically correct sentences".
- The **generation** of language can be more restrictive than "all grammatically correct sentences".

This can be shown as a diagram:



The boundaries of the actual language are not sharp, whereas the analysis and generation rules implemented on a computer are formal systems, whose boundaries (at least in principle) are well defined.

Analysis and generation are of course related to each other: they operate with the same concepts, and everything that is generated should also be analysable — as suggested by the inclusion of the innermost ellipse in the outermost one in the above picture. One technical way to achieve this is to start with a set of generation rules and perform analysis by **relaxing** these rules, or **smoothing** them. Smoothing can be performed both by adding new rules, by relaxing old rules, or by using grammar-external techniques such as machine learning. In this book, we will look at all these strategies.

But why write grammars at all — why not use machine learning (ML) all the way? The counter-question is: use for what? NLP tasks such as machine translation (MT) in large scale are mostly done by ML systems in these days, with Neural MT (NMT) as the dominating technology at the moment of writing this. Some scholars go as far as to claim that ML has made grammar obsolete (REF), or that the gradient nature of grammaticality shows that grammar is hopeless (Manning and Schütze, Lappin REF). One of the goals of this book is to meet this challenge and show how grammar can be useful in MT and other NLP tasks, both by itself and in various combinations with ML. Moreover, we will make a distinction between grammar as a descriptive and conceptual framework on one hand, and rule-based NLP systems based exclusively on grammar rules on the other. Even though such rule-based systems might be brittle and insufficient for many NLP tasks, a system based on machine learning can profit from using grammatical concepts instead of just surface-based data such as characted strings.

Most importantly, grammar is a scientific discipline valuable on its on right. Together with Euclidian geometry, it is perhaps the only branch of Ancient Greek science that is still mostly valid today; the Indian Panini tradition is likewise a vital part of science even today. Anyone who has studied the grammar of a language such as Ancient Greek, Sanskrit, Latin, Finnish, or Arabic — just to mention a few — is likely to be impressed by the depth of analysis performed by earlier scholars and bringing regularity to an extremely complex system, resulting in a beauty similar to mathematics. In fact, grammar at its best *is* mathematics with respect to rigour and explicitness (Lambek 1977).

Ignoring the tradition of grammar and the insights that can be learned from it is a suboptimal starting point for serious analysis of language. It is like approaching botany without knowing about Linnaeus's classification of plants.

On the other hand, making grammars *computational* helps keep them connected to the actual language. From the research point of view, a grammar is a **theory**, or at least a **hypothesis**, that should be **tested** against empirical data. Implementing a grammar on a computer makes it possible to test it on a much wider scale than what was possible for the grammarians of the past.

1.2 A first example

In this section, we will present a miniature explaining the concepts that will be thoroughly covered in Chapter 2. We will work out a single example of grammatical analysis, which, although a linguistic "toy example", illustrates a considerable amount of issues.

The example is the English sentence

The black cat sees us now.

The lowest level of analysis breaks this sentence down to **tokens**, which are sequences of characters separated by spaces. In addition, we introduce a space before the full stop from the end turn the first word to lower case. This is an example of a simple process od **tokenization**; in general, tokenization can be a much more complicated task than this. The result is a sequence of six tokens, separated by spaces:

```
the black cat   sees us   now .
```

The next level of analysis is **parts of speech tagging**, which classifies the tokens into **parts of speech** such as noun, verb, etc. We also leave out the full stop for the moment, although we could assign it "punctuation" as part of speech. The result looks as follows:

```
the black cat   sees us   now
DET ADJ   NOUN  VERB PRON ADV
```

We use here a standard set of part of speech tags from Universal Dependencies (UD,REF). Thus,

- DET = determiner
- ADJ = adjective
- NOUN = noun

- VERB = verb
- PRON = pronoun
- ADV = adverb

We will come back to the definitions of these concepts in Section 2.3.

Part of speech tagging can be completed with **lemmatization**, which specifies the **lemma** of each word, often called the **dictionary form** or **basic form**:

```
the black cat   sees us   now
DET ADJ   NOUN  VERB PRON ADV
the black cat   see  we   now
```

In addition to this, it can include **morphological analysis**, spelling out what **inflection form** each of the words appear in. For instance, *sees* is the "third person singular present indicative" of *see*, and *us* is the "accusative" of *we*. We will use **morphological tags** as abbreviations for these form descriptions:

```
the black cat   sees us   now
DET ADJ   NOUN  VERB PRON ADV
the black cat   see  we   now
_   Posit Sg    P3   Acc  _
```

The underscore _ means that the morphological tag is of no interest, because the word has only one inflection form. We will return to the classification of inflection forms in Section 2.4.

Our grammatical analysis has up to this point produced four levels of description: tokens, parts of speech, lemmas, and inflection forms. There are more levels to come, and it is convenient to change the format a bit, by rotating it 90 degrees from horizontal to vertical:

```
1 the    the    DET   _
2 black  black  ADJ   Posit
3 cat    cat    NOUN  Sg
4 sees   see    VERB  P3
5 us     we     PRON  Acc
6 now    now    ADV   _
```

```
1 the    the    DET   _     3  det
2 black  black  ADJ   Posit 3  amod
3 cat    cat    NOUN  Sg    4  nsubj
4 sees   see    VERB  P3    0  root
5 us     we     PRON  Acc   4  obj
6 now    now    ADV   _     4  advmod
```

Figure 1.1: A sentence analysed in a (slightly simplified) CoNLL format. The columns express pieces of information on each word — from left to right: token counter, surface word, lemma, POS tag, morphological tag, head, dependency label.

This format is a part of the grammatical description standard called CoNLL.[1] Each line in the CoNLL format contains one token and all information about that token. Its first element is a **token counter**, numbering the elements from 1 upwards.

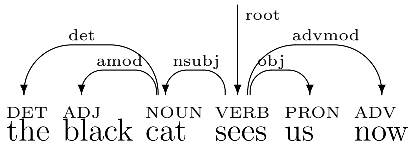
The counter in the CoNLL format is used for specifying relations between the words, and thereby to perform **syntactic analysis**. In syntax, we want to say, for instance, that *the* (token 1) determines *cat* (3), and that *black* is an adjectival modifier of *cat* (3). Such relations are expressed in the next columns of the CoNLL format, shown in Figure 1.1.

The tags shown in the last column are **dependency labels**, which specify the relation of each word to its **head**, which is the word whose position (token counter) is given in the second-last column. The tags used here belong, again, to the UD standard, and have the following meanings:

- `det`: determiner (typically of a noun)
- `amod`: adjectival modifier
- `nsubj`: noun subject (as opposed to sentential subjects)
- `root`: the "main word", with no head itself: its head is the "word 0"
- `obj`: object
- `advmod`: adverbial modifier

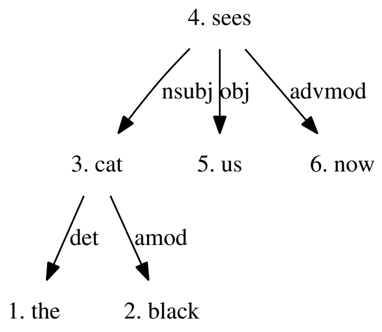
The CoNLL format is designed to be easy to process by a computer; the full format has some more columns than Figure 1.1, and the columns are separated by tab characters. CoNLL is also easy to convert to a more human-readable graphical format, known as **dependency trees**:

[1] Conference on Computational Natural Language Learning, Buchholz and Marsi 2006, https://www.aclweb.org/anthology/W06-2920.pdf



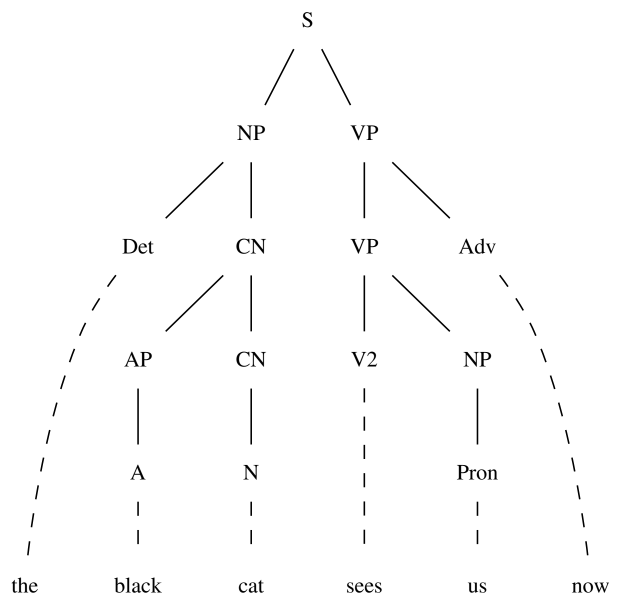
In this format, each word has an ingoing arrow from its head, marked by the dependency label. The POS tags are marked under the words. The lemma and the inflection form are usually not shown in this format: it is meant as a visualization, not as the ultimate data format.

Now, why is the depency tree called a "tree"? It is tree in the mathematical (graph-theoretical) sense, with the root word as root node, and each word branching to its **dependents**, that is, the words whose head it is. Here is a more tree-like visualization showing this clearly:



In this format, unlike the previous, the order of tokens cannot be read from the visualization, but has to be given explicitly.

For most of the third millennium, dependency trees have been the most popular form of syntactic analysis in computational linguistics. They have some properties that make them easy to process by machine learning, and we will return to these in Section REF. A more traditional kind of tree in linguistics is **phrase structure tree**, often called simply **syntax tree** or **parse tree**:



Here the words are grouped into **phrases**, which are grammatical units consisting of several words. Each subtree represents a phrase, and the nodes are **categories**, used for classifying phrase. Our example tree has the following **phrasal categories** (categories that can contain several subtrees and words):

- S = sentence
- NP = noun phrase
- VP = verb phrase
- CN = common noun phrase

The rest are **lexical categories**, which classify single words and hence just another term for parts of speech:

- Det = determiner
- A = adjective
- N = noun
- V = verb
- Pron = pronoun
- Adv = adverb

Phrasal categories appear in different places in the tree and contain phrases of different sizes:

- NP covers both *the black cat* and *us*
- CN covers both *black cat* and *cat*

- VP covers both *sees us now* and *sees us*

Phrases of the same category, such as the common nouns *black cat* and *cat*, satisfy an important property,

- **Substitution test**: phrases belong to the same category if can be replaced by each other in the same contexts *without loss of grammaticality.*

Let us try the substitution test by replacing the CN and VP parts with shorter phrases:

the cat sees us now

is indeed a grammatical sentence just like the original. However, if we swap the NPs *the black cat* and *us*, we get

us sees the black cat now

which is not grammatical (at least if *us* is regarded as the subject as in the normal English word order). What is happening here?

The answer is that the substitution test still works if we apply it in a generalized way, by using different inflection forms of the phrases:

we see the black cat now

The choice of forms must obey the rules of **agreement** of English, so that we select proper forms for each lemma. The substitution test in fact means that *lemmas* of the same category can be replaced with each other, but their *forms* must be changed to the forms valid for the original place of occurrence. What is more, the forms of some other words may have to change as well. This is what happens to the verb *see* in our example when the subject NP is changed.

Defining agreement properly is one of the most challenging (and interesting) tasks for a computational grammarian. We will return to it in Section 5.1. Until them, we can be satisfied with grammars that do not implement agreement: when we are analysing language rather than generating it, we do not need to care about the absolute correctness of the input.

1.3 The interlingual perspective

Talking about lemmas as opposed to concrete word forms is a **linguistic abstraction**. Like in all sciences, abstractions are needed in linguistics to understand the object of study beyond the immediataly obvious — in this case, how the substitution test works. Abstractions come hand in hand with **generalizations**, which enable **predictions**. For instance, a verb may have some form that has never been observed even in a corpus of internet size before, but a competent speaker can immediately understand that form because it follows from the generalization that all verbs and inflected in a certain way. Grammatical generalizations can be contrasted to a "purely empirical" approach to linguistics (or any other discipline), which only accepts data that has been seen before. Such an approach was advocated in the Renaissance time by scholars who claimed that only the expressions that could be found in the writings of classical Roman authors could really count as Latin (REF). In modern days, similar views can sometimes be seen connected to data-driven approaches (REF).

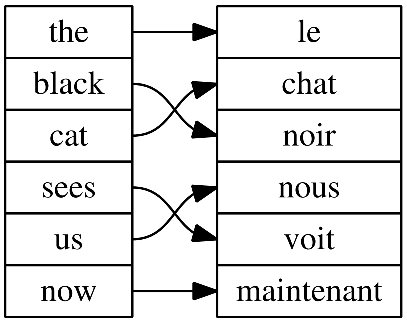
Abstractions that apply across languages are a particularly powerful generalization, sometimes made under the title of **universal grammar**. In this book, while we start modestly by looking at individual languages in separation (Chapter 2), we will take the step to **interlingual abstractions** from Chapter 3 on. The interlingual perspective is a fruitful way to understand relations between languages — both similarities and differences. This is less bold than the claim sometimes associated with universal grammar, according to which all languages have exactly the same grammar. For us, it is enough to say that the same concepts and structures apply to different languages, not that they apply exactly the same way to all languages.

Let us start with the words in our simple example. If we ask how the English verb *see* translates to French, anyone who knows a bit of these two languages would say *voir*. However, this answer is not accurate when it comes to the different forms of the verb. Thus the form *see* itself translates differently for *I see* (*je vois*), *we see* (*nous voyons*), the infinitive *see* (*voir*), and so on. In fact, the English verb has only 5 different forms, whereas the French verb has 51. Nevertheless, it makes perfect sense to say that *see* translates to *voir*, when we talk on the abstraction level of lemmas.

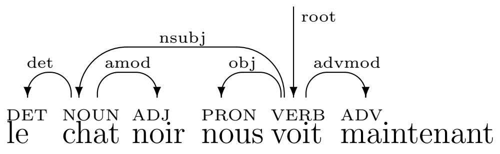
Another abstraction level that we need to relate languages with each other is **syntactic structure**. When we translate the English sentence to French, we get

le chat noir nous voit maintenant

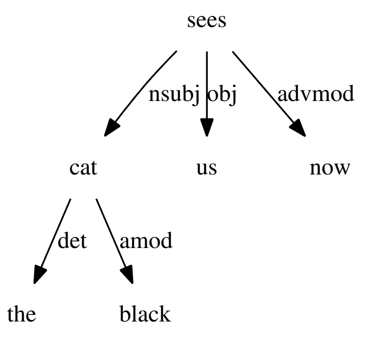
The correspondences between the words are shown in the following **word alignment** diagram:

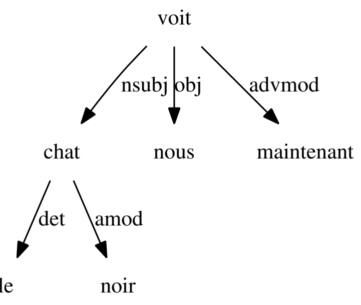


The diagram shows that the words for *black* and *cat* are swapped, as are *sees* and *us*. However, on the level of syntactic structure, we can still see the same dependency labels relating those words: *noir* is an adjectival modifier of *chat*, just like *black* is of *cat*, and the same applies to the verb and the object:



This can be seen even more clearly if we use the tree-like dependency format and drop the token numbers. Then we get two trees with exactly the same form, except that the words are different:





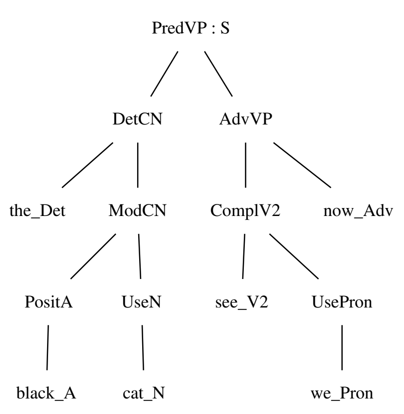
What makes UD *universal* is precisely that it uses the same grammatical relations for different languages. Thus it is possible to impose the same syntactic structure on different language where the words are different and appear in different orders. This structure is a linguistic abstraction on the level of syntax in a similar sense as lemmas are abstractions on the level of words; notice that we have also abstracted from the inflection forms here.

The **interlingual perspective**, which this book is about, means that we look for linguistic abstractions that capture the essential features of equivalent expressions in different languages. This "essence" is the common thing that translation tries to preserve. It is different from **semantics** in the usual sense: correct translation should not only preserve semantic meaning but also express the meaning "in the same way". Lemmas and UD dependency trees approximate the essence of expression by abstracting away of "irrelevant" language-dependent details. We will go one step further and introduce **abstract syntax trees** (AST).

An AST is, in a sense, a synthesis of the information contained in dependency trees and phrase structure trees. It groups together words into phrases, just like phrase structure trees do, but represents the words by their lemmas rather than their inflected forms. Each grouping (subtree) is labelled by a **construction function**, from which one can compute the heads and dependency labels of each word. What is more, each construction function is equipped with a **linearization rule**, which tells how it is converted to a sequence of words with correct inflection forms in correct format.

The AST for our example sentence is shown in Figure 1.2. The same tree covers both the English and the French sentence, via linearization rules for those languages. Each node shows the category (as in the phrase structure tree) and the construction function (a concept proper to AST). The branches under each construction function are decorated by UD dependency labels. However, only the construction functions are strictly speaking necessary, since both the categories and the dependency labels can be deterministically computed from them. Construction functions are ways to combine AST units to larger units. The functions in Figure 1.2 have the following meanings:

- `PredVP`, **predication**, combines an NP and a VP into a sentence (S)
- `DetCN`, **determination**, combines a Det and a CN to an NP
- `ModCN`, **modification**, extends a CN with an AP
- `PositA`, positive form selection, forms an AP from an A
- `UseN`, using a noun, forms a CN from a single N
- `AdvVP`, **adverbial modification**, extends a VP with an Adv
- `ComplV2`, **complementation**, gives an NP complement (such as object) to a V2 (two-place verb)
- `UsePron`, using a pronoun, forms an NP from a single Pron



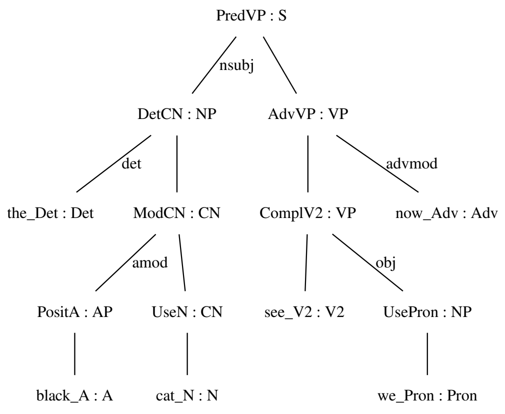


Figure 1.2: Abstract syntax tree in its original form (left) and redundantly decorated with categories and dependency labels (right).

- `the_Det, black_A, cat_N, see_V2, we_Pron, now_Adv` are construction functions representing single lemmas

What about the linearization rules? Anticipating the proper introduction in Chapter 4, let us just so the rule of adjectival modification, which in English poses the AP before the CN, in French after it:

```
lin ModCN ap cn = ap ++ cn  -- English
lin ModCN ap cn = cn ++ ap  -- French
```

These rules do not yet express the agreement in number and (in French) gender, but just the order of **concatenation** (marked ++).

The functions and categories shown in Figure 1.2, together with a number of others, have been successfully used to analyse over 40 languages from different language families in the GF Resource Grammar project. Their computer implementation enables automatic structure-preserving translation between the languages. The translation proceeds by **parsing** (analysing) the source language input as an AST, and linearizing the AST into the target language. The term **interlingua** is thus appropriate for the AST, connecting it to the tradition of interlingual translation.

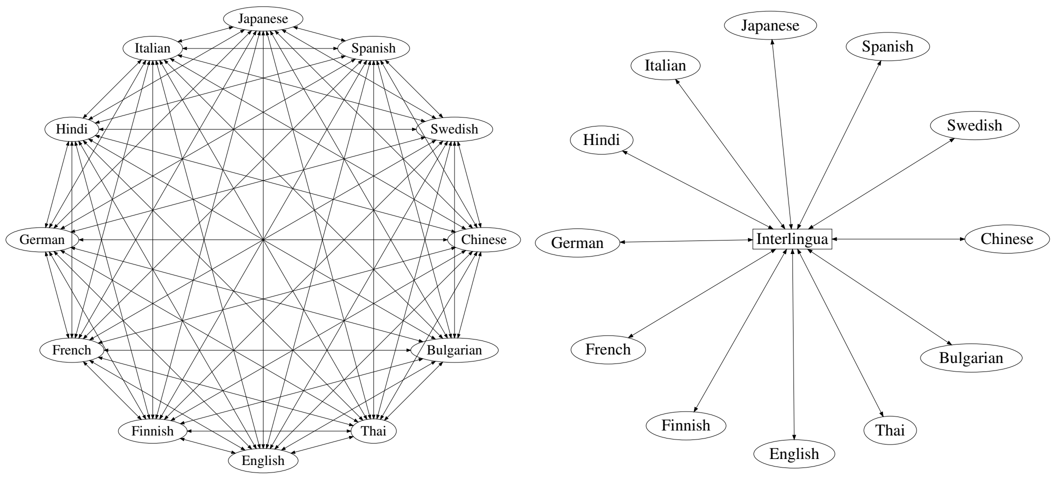
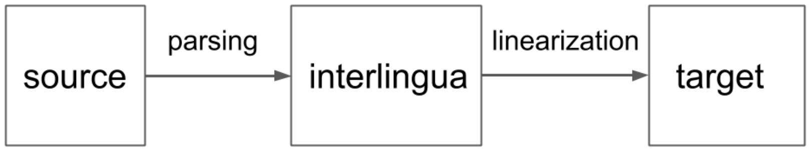


Figure 1.3: A translation system with 12 languages using transfer (left) vs interlingua (right).



An interlingual system can work for many languages simultaneously, as shown in Figure 1.3. The practical advantage is that one does not need to build translation functions for pairs of languages, but just between each language and the interlingua. For the 12 languages of Figure 1.3, this means $24 = 2 \times 12$ functions, instead of $132 = 12 \times 11$.

What should there be in the interlingua? Ideally,

- An interligua should express everything that is needed for faithful trans-

`le`	`chat`	`noir`	`nous`	`voit`	`maintenant`
`le`	`chat`	`noir`	`nous`	`voir`	`maintenant`
`DET`	`NOUN`	`ADJ`	`PRON`	`VERB`	`ADV`
`MascSg`	`Sg`	`MascSg`	`PlP1`	`PresSg2`	`_`
`il`	`gatto`	`nero`	`noi`	`vedere`	`adesso`
`il`	`gatto`	`nero`	`ci`	`vede`	`adesso`

Figure 1.4: Word-by-word translation from French to Italian. Each lemma is replaced by its translation equivalent, and the same inflection form is selected for it.

lation.

This clearly includes syntactic relations, such as subject and object, to get the agreement and word order right. But it also includes features that are normally taken as belonging to **semantics** rather than syntax. Thus words should be analysed not only to lemmas but to **word senses**. For instance, English *drug* in the sense of medicine has in most languages a different translation from *drug* as narcotic substance, and the interlingua should specify which of the senses is meant.

Word senses and many other semantic features can be expressed in an AST. But if we just look at the problem of translation, a more shallow grammatical analysis can be sufficient, at least in special cases where we don't attempt perfect translation or if the languages are similar to each other. The former perspective is standardly taken in main-stream MT systems, which are meant to be helpful but not replace professional translation. The latter perspective is presented by Apertium (Forcada et al. 2011), which is a system for translating between closely related languages in a word-by-word fashion. The algorithm is briefly:

- for each word, find its POS tag, lemma, and morphological features
- look up the lemma and its POS tag in a dictionary
- render the lemma by using the same inflection forms as in the source

Our running example illustrates the procedure when translating from French to Italian, in Figure 1.4. The result in Figure 1.4 is a perfectly correct translation.

However, even closely related languages may have differences that cause translation errors with this method:

- Words can have different genders, which means that the agreement of other words fails: French feminine *la méthode*, Italian masculine *il metodo* ("the method").

- Word order can differ: French *le chat veut nous voir*, Italian *il gatto vuole vederci* ("the cat wants to see us", order "us see" in French, "see+us" in Italian without a space)
- Number of words can differ: French *je veux le voir*, Italian *voglio vederlo* ("I want to see him", Italian drops the subject pronoun and glues the object with the verb)

The Apertium system itself has ways to cope with some of these problems, but the examples illustrate the fact that simple word-by-word tranlation is not perfect even for closely related languages.

Chapter 2

Grammatical analysis: words

In this and the following two chapters, we take a a systematic and comprehensive look at grammatical analysis on different levels. We will first assume that the analysis is performed **manually**, that is, by an intelligent human rather than a computer. The ability of manually analyse language is essential for both rule-based and (supervised) machine learning approaches:

- To enable machine learning, someone has to **annotate** the linguistic data with information such as part of speech tags and syntactic relations.
- To write grammatical rule systems, the grammarian must understand what the rules are expected to produce.

The chapters will conclude with simple techniques for automating the analysis by the use of morphological lexicon (this chapter) and grammar rules (Chapter 4). Details about the algorithms that enable automation are given in Chapter refalgorithms.

This chapter is divided into following sections:

- Section 2.1 discusses what size of units grammatical analysis should cover; the traditional answer is "sentences", but this is not a fully satisfactory answer.
- Section 2.2 discusses how text is divided to words and, more generally, to tokens.
- Section 2.3 introduces the different parts of speech (lexical categories) and how words are classified into them.
- Section 2.4 introduces morphological features and inflection, as applicable to different parts of speech.
- Section 2.5 gives guidelines for building a morphological lexicon, which

can be used for recognizing all forms of all words in a language.
Throughout this chapter and the next, we will use the notation of Universal Depencies (UD), which is designed to be applicable to all languages. The UD names for parts of speech, morphological features, and syntactic relations have become a widely used standard even outside the UD programme itself. One advantage of them is that they are easy to understand for anyone familiar with grammatical concepts — for instance, that adjectives are called `ADJ` and not for instance `JJ` as in the other popular tagset of Penn treebank (REF).

2.1 Units of analysis

Before starting the analysis, we need to decide what exactly we are analysing:

- What are the units of language that we are analyse in grammar?

The usual answer is **sentences**, and the first step in analysis must therefore be the division of a text into sentences. The baseline algorithm for this is to split up the text by the following definition:

- A sentence starts with a capital letter and ends with a punctuation mark, one of ".?!".

While this might work for 90% of sentences of English, there are exceptions, such as the full stop used in abbreviations:

> *Many countries, e.g. Denmark, have closed their borders to stem the spread of the virus.*

A human reader would not split this sentence between *e.g.* and *Denmark*, but the simple-minded algorithm would.

A more linguistic definition of a sentence says that it should contain a **main verb**. However, if this definition is adopted, a sentence is no longer an adequate unit of grammatical analysis. For example, the title of this book would be ruled out, since it contains no verb.

A more general linguistic unit is called **utterance**, which is intuitively defined as any sequence of words that could be uttered in a dialogue, or used independently in communication e.g. as a title. An utterance can thus be a sentence, a noun phrase (like typically a title), an interrogative (like *why* in a dialogue), an answer to a question — actually, almost any grammatical phrase. It might not even be a phrase in a grammatical sense, but just something that is used for communication. For instance, a road sign could say

Oslo 291

to indicate the distance to Oslo, and this text has its own structure and meaning, which are interesting to analyse.

Utterances are often "shorter than complete sentences", e.g. by lacking a verb. However, from the interlingual perspective we can also be interested in units that are longer. The adequate unit of analysis is then a **translation unit**, and the analysis must find out everything that is needed for translation. Question-answer pairs in a dialogue are a case in point:

What language do you speak?
Swedish.

What is your nationality?
Swedish.

The word *Swedish* is in the first dialogue a noun phrase standing for a language. In the second dialogue, it is an adjective. In most languages, these are two different words, and faithful translation should hence treat the question-answer pair as a unit.

In the rest of this chapter, we will mostly assume the "utterance" as the unit of grammatical analysis. This is what is done in most NLP systems, including MT systems and UD parsing. We will not try to define the notion of utterance precisely, and even less to automate the splitting of text to distinct utterances.

However, the full picture of analysis has to extend from utterances to **text**, which consists of an arbitrary number of utterances, including dialogues by many participants. This is needed for instance in the translation of pronouns between languages that have different gender systeams. Thus English *it* has three translations in German: *er* (masculine), *sie* (feminine), *es* (neuter). The choice depends on the **referent** of the pronoun, which can be given elsewhere in the text, or even outside the text. We will return to units larger than utterances in Section 8.4

2.2 Tokens

Assuming that we have identified the utterances, the next step is to split them into tokens. For languages using an alphabet such as Latin, Greek, or Cyrillic, the following procedure comes a long way:

1. Introduce a space on both sides of every punctuation mark (one of .!?.:; and some more).
2. Lower-case the first token, unless it is a proper name or similar (given in a special list).
3. Tokens are now the sequences of non-space characters separated by one or more spaces.

Here is an example:

```
The cat sees John, but he does not see the cat.
the cat sees John , but he does not see the cat .
```

An obviously difficult step is (2): exhaustive lists of proper names and other capitalized words are not always available. The result can moreover be ambiguous: *United* can be the participle *united* or also the name of a football club. To find out which, a deeper grammatical analysis is needed, and the tokenization may postpone the decision by leaving alternatives.

Another difficulty is punctuation marks that do not separate tokens, such as in *e.g.* and *11:30*.

Depending on language, the main difficulty may be tokens not separated by spaces or punctuation marks. A ubiquitous example is **clitics**: small, typically unstressed words, which are are often glued to other words. We saw in Section 1.3 the Italian clitic *ci* ("us"), which is sometimes a separate token, sometimes glued to another word:

ci vedi "you see us"

vuoi vederci "you want to see us"

In English, the genitive ending *'s* might be treated as a clitic, but it is easy to recognize by the apostrophe. The only problem comes on the POS tagging level, when it must be distinguished from abbreviated *is* and *has*.

The abbreviated negation *n't* is more tricky. In addition to tokenization, it may undergo **normalization** into the word *not*. There are at least three different situations:

- straightforward introduction of token boundary: *hasn't* tokenized as *has n't*, perhaps further normalized to *has not*
- token boundary with change of a token: *can't* tokenized as *can n't*, *won't* as *will n't*
- word order order change after normalization: *hasn't he* normalized to *has not he*, converted to more grammatical *has he not*

Other clitics in English are *'ll*, *'m*, *'re*, *'d* (ambiguous between *would* and *had*), *'ve*.

The UD guidelines for tokenization are separate for each language, and the practice may also vary from one language to the other. The English guidelines can be found in

`https://universaldependencies.org/en/index.html`

In this book, we will most of the time just assume that tokenization is performed in some way that is consistent with the later phases of analysis. However, the problem cannot be simply ignored in those languages whose script does not indicate word boundaries, such as Chinese or Thai. As we will see in REF, it can then be better to leave tokenization to a later phase of analysis, for instance, to syntactic parsing that decides on word boundaries on the basis of what is possible to parse.

2.3 Parts of speech: an overview

Assuming an utterance split into tokens, the next step is to assign a part of speech tag to each token. This task has two steps, where the latter cannot be fully separated from syntactic analysis:

- **Morphological analysis** tells us that a word W can be form F of lemma L in category C. The typical outcome is many analyses.
- **Part of speech disambiguation** determines which morphological analysis is the correct one.

An example of ambiguity is the English token *bears*, with the analyses

```
bears bear N Plur
bears bear V PresSgP3
```

The main disambiguation techniques are

- rule-based: exclude analyses that do not fit syntactically correct sentences;
- statistical: select analyses that lead to the most probable sequences of tagged words.

For example, if we have the following ambiguity,

```
bears       are         shy
N Plur      V PresPlur  A
V PresSgP3
```

we might have a rule-based analysis that recognizes the sentence where the first word is a plural noun, but no rule where that fits when it is a verb. A statistical analysis, on the other hand, would look at a corpus of previously POS-tagged data and notice that the combination N-V-A is much more common than V-V-A.

The POS tagging phase can in principle be performed as a part of the syntactic analysis proper, but it is often performed as a light-weight preprocessing by using statistics or heuristic rules. We will here simply that a human can do it accurately by using her knowledge of syntax and semantics, and look at the possible ways to automate it Chapter 10.

The plan for this section is hence to go through the different parts of speech and explain what words belong to them. We will start with the **Universal Tag Set** used in UD treebanks and listed in

`https://universaldependencies.org/u/pos/index.html`

Table 2.3 is based on this list, with example words taken from English PUD (Parallel UD) treebank. The table shows several examples that occur with many POS tags, for instance, *well*.

The Universal Tag Set overlaps partly with other classifications of words that appear in other computational approaches and grammar books. The essential thing is not the tags themselves, but the grammar terms and how they are defined: how do we select the correct classification of a word?

Linguistic tradition uses three different kinds of criteria for classifying words (cf. Lyons 1967):

- **Semantic**: what kind of things does the word stand for?
- **Syntactic**: what syntactic combinations is the word used in?
- **Morphological**: what inflection forms and morphological features does the word have?

Semantic criteria are common in popular grammar descriptions. There one can read for instance that "nouns stand for objects, adjectives for qualities, and verbs for actions". But they are uncommon in computational approaches, because their application is hard to decide by the computer. However, they can still be relevant in computational semantics: the question "what kind of things" is then answered by a **logical type**, which is a formalized classification of semantic objects. We will return to this notion in Section 8.1.

The syntactic criterion is can be seen as a special case of the **substitution test** for phrases, introduces in Section 1.2: words belong to the same

tag	grammar term	examples
`ADJ`	adjective	*big, other, particular*
`ADP`	adposition	*after, on, within*
`ADV`	adverb	*now, again, apparently, well*
`AUX`	auxiliary	*have, should, will*
`CCONJ`	coordinating conjunction	*or, and, but, either*
`DET`	determiner	*a, the, this*
`INTJ`	interjection	*yes, well, hmmm*
`NOUN`	noun	*car, quality, well*
`NUM`	numeral	*one, two, billion, 400, 30.00, 9:30*
`PART`	particle	*not, to, 's*
`PRON`	pronoun	*I, it, my, who*
`PROPN`	proper noun	*Ahmed, September, San*
`PUNCT`	punctuation	*. , ! ? : ; ( ) " -*
`SCONJ`	subordinating conjunction	*that, because, after*
`SYM`	symbol	*$ % + / :)*
`VERB`	verb	*find, know, be, have*
`X`	other	*etc, eg*

Table 2.1: Universal part of speech in UD standard 2, together with examples.

lexical category if they can be substituted for each other without loss of grammaticality.

Syntactic and morphological criteria are tightly connected, especially in languages with rich morphology. As a general rule,

- The morphological features of each lexical category match with the agreement rules in syntactic combinations.

A typical example is nouns and adjectives in languages like French and Italian:

- syntactically,
    - adjectives are words that modify nouns
    - adjectives agree to nouns in gender and number
- morphologically,
    - nouns inflect for number and have an inherent gender
    - adjectives inflect for both number and gender

The interplay of morphological inflection and syntactic agreement makes grammar writing into a complex puzzle. At the same time, it creates much of the beauty and coherence of computational grammar, and can be described in compact and elegant ways if proper formal means are used.

Morphological inflections are largely language-specific: in English, for instance, nouns have no gender, and adjectives are inflected for neither number nor gender. In some languages, e.g. Chinese, morphological POS definitions are hardly applicable at all, as there is no inflection. On the other hand, syntactic combinations are largely universal: adjectives can in all languages be used for modifying nouns. In the interlingual perspective, we can sometimes best identify "adjectives" and "nouns" by looking for translation equivalents of words that have clear morphological indicators in some other language.

Furthermore, some generalizations can be made about the assignment of morphological features to parts of speech, of the form

- if a language has feature F, then words in category C tend to have F

For example,

- if a language has cases, then nouns are likely to be inflected for case.

This principle is not watertight, but it can be a good guideline when approaching a new language. Again, what makes it work is the interplay with agreement in syntactic combinations: nouns are typically used as subjects and objects of verbs, and these positions are marked by different cases.

Because of the importance of syntactic criteria in the classification of words, we will postpone its detailed discussion till after introducing both morphological features and syntactic relations.

Yet another important distinction among words is between **content words** and **function words**. Content words are easy to list: nouns, adjectives, verbs, adverbs, interjections. Languages typically have thousands of each of them, and they are moreover **open classes**, which means that new content words are often added to the languages. Function words are divided into many more categories, such as pronouns, determiners, conjunctions, prepositions, but these are **closed** classes with very few words in each, and new words rarely added. But if we apply the morphological and syntactic criteria rigorously, we will end up with a large number of classes, whose words behave differently either in morphology or syntax or both.

2.4 Morphological features and inflection

The inflected forms of a word can be collected to an **inflection table**. The table allows us to look up the forms for any given features, and also, maybe less quickly, the features belonging to any given form. The most well-known inflection table is probably the one for the Latin noun *rosa* ("rose") or some other similar noun:

	singular	plural
nominative	*ros**a***	*ros**ae***
accusative	*ros**am***	*ros**as***
genitive	*ros**ae***	*ros**arum***
dative	*ros**ae***	*ros**is***
ablative	*ros**a***	*ros**is***

Here are some general conventions used in inflection tables:

- A table operates on a number of **features**, here on number (singular or plural) and case (listed on the leftmost column).
- The word forms are written in italics, often with **endings** marked in boldface; the rest of the form is the **stem** of the word.
- Some forms appear in several cells of the table: these are known as **syncretic** forms.
- The table can be applied to other words by just changing the stem — for instance, from *ros* to *mens* (*mensa*, "table").

The last-mentioned feature, changing the stem, makes the table usable as a **function** where the stem acts as variable. A function producing the inflection table of a word is called a **paradigm**. Paradigms for nouns and adjectives are traditionally called **declensions**; for verbs, they are called **conjugations**.

The paradigm shown above is known as "the "first declension" of Latin nouns. Traditional grammars distinguish between five declensions in Latin, but these do not exactly match all the different functions there are for inflecting Lating nouns. Thus for instance the "second declension" has three variants (for nouns ending in *us*, *er*, or *um*). There is also a considerable number of nouns that do not fit the usual paradigms, and are therefore called **irregular**.

Some computational approaches to paradigms introduce a separate paradigm for every different set of endings that can be attached to a stem. Thus for instance Swedish nouns, which traditional grammarians classify into five declensions (no doubt inspired by Latin), are given as many as 345 (CHECK) declensions in the first computational morphology of Hellberg (1976, REF).

While strinct stem+lemma based paradigms are mechanically applicable and more accurate than the traditional five declensions, they can result in unintuitive analyses. For instance, English noun paradigms would then include

fly* — *flies

where the "stem" is *fl*. This description can be avoided by letting the paradigms perform predictable variations in stems and endings. Thus the English plural *s* can trigger the predictable variations shown in *fly-flies* and *baby-babies*, but also in *kiss-kisses* and *bush-bushes*. We will call such inflection functions **smart paradigms**.

A prerequisite for paradigms to work is that

- all words in the same lexical category have the same morphological features.

This is actually the essence of the morphological definition of parts of speech. Thus for instance *all* Latin nouns inflect for number and case, and the first declension (or some of the other four) can be applied to them. This is a strong generalization, which makes senses even if it seems to be easy to find counterexamples:

- Some nouns don't have all the forms — they might for instance appear only in the plural (these are known as **plurale tantum** nouns).
- Some nouns are also inflected for gender (e.g. *imperator* "emperor" (masculine) *imperatrix* "empress" (feminine)).
- Some declensions (the second) also have a sixth case, the vocative.
- Some forms are the same for all words (the dative and ablative plural).

Similar objections apply to most languages, e.g. to English with plurale tantum nouns such as *scissors*, as well as feminine-masculine pairs such as *emperor-empress*. It is up to the grammar writer to choose one of the possible solutions:

- to split a class to several ones, each of them defined by what forms there are in the inflection table (this makes sense for frequent classes such as plurale tantum);
- to permit empty slots in an inflection table (this makes sense for single words accidentally missing some forms);
- to redefine the table layout in a non-traditional way, so that for instance dative/ablative distinction in Latin only applies to the singular.

The layout of the table specifies what inflection forms there are in the table, not how they are built. We will use the term **inflection type** to denote this layout. It is similar to the notion of **datatype** in programming languages — in fact, as we will see, it can be accurately defined as such a type when the grammar is formalized, even in complicated cases such as missing dative/ablative distinction in the Latin plural (Section˜\ref{parameters}).

The morphological criterion of parts of speech can thus be partly defined as the inflection type. Another part is **inherent features**. These are morphological properties that a word "just has", instead of being inflected for them. For nouns in Latin (as well as French, Italian, Swedish, etc), gender is an inherent feature. The Latin noun *rosa* has the inherent gender feminine, and there is no "masculine form" or "neuter form" of *rosa*. In contrast to this, adjectives in Latin (as well as French, Italian, Swedish, etc) have gender as an **inflectional feature**, in addition to the features belonging to nouns. The "logic" of this comes from syntax: adjectives have to agree in gender to the nouns that they modify. Since the noun cannot change its gender, it is the adjective that has to comply.

The morphological definition of a lexical category (a.k.a. part of speech) can now be given as a combination of two things:

- inflectional features, i.e. what features the words in this category are inflected for;
- inherent features, i.e. what features the words have as fixed properties.

Inflection types and inherent features are built from the same set of features. What features there are, and what are their possible values, depends on language. Table 2.4 gives a list of the most common features and some of their values, using the UD terms. The source is

feature	typical values
`Number`	`Sing`ular, `Plur`al, `Dual` (in e.g. Arabic)
`Gender`	`Masc`uline, `Fem`inine, `Neut`er, `Com`mon (in e.g. Swedish and Dutch)
`Person`	1, 2, 3
`Case`	`Nom`inative, `Acc`usative, `Gen`itive, `Erg`ative
`Degree`	`Pos`itive, `Cmp` (comparative), `Sup`erlative
`Tense`	`Past`, `Pres`ent, `Fut`ure
`Mood`	`Ind`icative, `Imp`erative, `Sub`junctive
`Aspect`	`Perf`ect, `Imp`erfect, `Prog`ressive
`Voice`	`Act`ive, `Pass`ive

Example:

`Mood=Ind|Number=Sing|Person=3|Tense=Pres`

"third person singular present indicative"

Table 2.2: Some morphological features in in UD standard 2, together with their typical values, and an example of a form description. The value names are usually prefixes of full terms, here distinguished by the use of typewriter font.

`https://universaldependencies.org/u/feat/index.html`

As shown in Table 2.4, form descriptions in the UD standard are lists of feature=value pairs separated by vertical bars and sorted alphabetically. We will not follow this convention everywhere in this book, because of its verbosity, but we will use the standard feature and value names whenever possible.

The traditional term for morphological features such as number and gender is **grammatical category**. We will, however, avoid this term, since we follow the modern linguistic use of the term "category" for classes of words and phrases.

Depending on language, the morphological definition of parts of speech can be more or less distinctive. In Chinese, it is practically useless. In English, it takes us some way, distinguishing between

- nouns, with inflectional number (as well as case, if the *'s* genitive is counted)
- adjectives, with inflectional degree

- verbs, with inflectional tense and (to some extent) number and person
- personal pronouns, with inflectional case (*I*, *me*, and the possessive *my* if counted) and inherent number and person

However, it also leaves us with many classes with no inflection: adverbs, conjunctions, prepositions, etc. To distinguish these classes from each other, we need to apply syntactic or semantic criteria.

2.5 Morphological lexicon

The largest part of a computational grammar system is usually the **lexicon**, which gives information about the words of the language. Natural languages may have up to several hundred thousand lemmas, of which some 50,000 can be in active use. The total number of word forms can be millions, as for instance Latin verbs have hundreds of forms. Some word forms might never appear in a given corpus of texts (even if the corpus is the whole internet), but still "exist" in the sense of being predicted by the paradigms and easily producible by competent speakers when requested.

From the computational grammar point of view, there are two important requirement for the morphological lexicon:

- Efficiency: it should enable fast analysis and synthesis of word forms.
- Effort: it should be constructible and extensible by the minimum of work.

The implementation techniques and examples presented in this book enable analysis speed of millions of words per second on a modern laptop computer, so efficiency in the sense of speed should not be an issue. However, the size of the lexicon can be a bigger problem, which can also affect the speed if for instance the computer's memory is small. limited. The main efficiency issue we shall consider here is hence the size of the lexicon.

The effort requirement covers both the manual construction and machine learning of lexical resources. A good approach to both is to enable building every lexical entry from the minimal amount of information. Traditional dictionaries are often presented in this way. To define the word *rosa* in a Latin dictionary, it is not necessary to to show the entire inflection table, but just a part of speech tag ("n" for nouns), a reference to the paradigm ("I" for the first declension), and, in the case of nouns, the inherent gender (because some nouns of the first declension are masculine):

rosa n.f. I

An alternative, often used in Latin dictionaries, is to give the nominative and genitive singular forms:

> *rosa*, *-ae* n.f.

This practice is based on the fact that these two forms almost always determine the declension and thereby all the other forms.

Most dictionaries follow the latter practice, where each class of words has a conventional list of **characteristic forms** that determine all other forms. Another example is English verbs: for regular verbs, it is enough to give the infinitive

> *walk* v.

The reader is assumed to be able to expand this to the list with regular endings:

> *walk* (infinitive), *walk**s*** (3rd person singular present), *walk**ed*** (past indicative and participle), *walk**ing*** (present participle)

Competent dictionary users may be assumed also to apply the regular verb paradigm to cases where predictable stem and ending variations are needed. Here are some examples, where the arrow marks the point from which the reader is assumed to produce the full forms:

> *wash* v. $\longrightarrow$ *wash, washes, washed, washing*
>
> *use* v. $\longrightarrow$ *use, uses, used, using*
>
> *cry* v. $\longrightarrow$ *cry, cries, cried, crying*
>
> *stop* v. $\longrightarrow$ *stop, stops, stopped, stopping*

The last example, final consonant duplication, is not always predictable from the infinitive, in which case the dictionary may give a second form:

> *omit, omitted* v. $\longrightarrow$ *omit, omits, omitted, omitting*
>
> *vomit, vomited* v. $\longrightarrow$ *vomit, vomits, vomited, vomiting*

The most obvious case of unpredictable inflection is so-called irregular verbs, which are specified by three characteristic forms:

> *sing, sang, sung* v. $\longrightarrow$ *sing, sings, sang, sung, singing*
>
> *sit, sat, sat* v. $\longrightarrow$ *sit, sits, sat, sat, sitting*

The only verbs in English where three characteristic forms are not enough are

- *have* (with the 3rd person singular present *has*)
- *be* (with eight different forms)
- auxiliary verbs with special negations (such as *can't, cannot* of *can*).

What is more, no verb other than *be* and the auxiliaries has more than five different forms. These five forms are thus all that have to be stored in the inflection tables of verbs. The morphological lexicon should not generate compound "forms" such as *has walked*, or distinguish between *walk* as infinitive and the five present tenses (1st and 2nd person singular, all persons plural). These distinctions are of course needed in *syntax*, to define subject-verb agreement, but they do not belong to *morphology*, because they never result in different word forms.

A lexicon giving the "full picture" of the words of a language should also indicate

- **Sense distinctions**: many words have different senses, which may have different translations in other languages.
- **Valencies**: a verb may be used as transitive as well as intransitive, and in different combinations with complements, prepositions, and particles, and these uses may have different senses translations; the same applies to some nouns and adjectives as well.

These aspects are of essential on higher levels of grammatical analysis, in particular from the interlingual perspective; we shall return to this task in Section 5.6. But the morphological lexicon has its own function, and it should avoid all redundancy: it should

- Minimize the number of forms stored for each class of words.
- Minimize the number of forms that the author of the lexicon needs to give.
- Minimize the number of entries, so that each combination of lemma, POS, inflection table, and inherent features occurs only once.

These requirements help to keep the size of the lexicon to the minimum. At the same time, they are consequences of the DRY principle, which applies to all program design:

- DRY: Don't Repeat Yourself.

Let us summarize the design principle with a concrete example from English. The word *lie* is both a noun and a verb, and the verb has two paradigms — the regular *lie-lied* and the irregular *lie-lay-lain.* This means we need three entries in the morphological lexicon:

lie n. $\longrightarrow$ *lie, lies*

lie v. $\longrightarrow$ *lie, lies, lied, lied, lying*

lie, lay, lain v. $\longrightarrow$ *lie, lies, lay, lain, lying*

The verbs — the irregular one in particular — have several valencies and combinations, such as *lie down*, *lie ahead.* A single valency pattern can have different senses, such as "be in a lying position" vs. "be located somewhere". All of these will need to have different entries in an **interlingual translation lexicon**. But that lexicon should not repeat the morphological information, but inherit it from the morphological lexicon.

We will return to the actual implementation of morphological lexicon in Section 5.2, where we use GF as a tool for both building the lexicon and using it for analysis and synthesis. However, we can already come a long way in NLP with a very simple format of lexicon: the **full-form lexicon**, which lists are lemmas with their POS tag, inflection forms, and inherent features. The forms are given in a fixed order, in which each position corresponds to a specific form description. A full-form lexicon entry might look as follows:

```
rosa N F rosam rosae rosae rosa rosae rosas rosarum rosis
```

You might want to compare this with the inflection table given in the previous section. Then you can notice that the last form represents both the dative and ablative of plural: since these are always the same, storing them both in the full-form lexicon would be redundant.

A typical file format for a full-form lexicon is one entry per line, forms separated by spaces or tabs. More sophisticated formats such as XML or JSON are of common as well and have some advantages such as standard programming tools.

The generation of forms from a full-form lexicon is made by looking up the lamma and POS tag, and picking the relevant position on the same line. For example the genitive plural in the Latin example is the tenth word of the line. Morphological analysis can be performed by collecting all word forms of the lexicon in a search tree or a finite-state automaton.

Chapter 3

Grammatical analysis: dependencies

In this chapter, we extend grammatical analysis from words to utterances. We will continue with the assumption that the analysis is carried out manually, and focus on giving the intuition for how to do this.

- Section 3.1 lists the most important syntactic relations in Universal Dependencies (UD).
- Section 3.2 takes a detailed look at the relations between the root of a clause and its dependents.
- Section 3.3 takes a detailed look at the relations involving words in nominal categories.
- Section 3.4 takes a detailed look at subordinate clauses.
- Section 3.5 takes a detailed look at coordination structures.
- Section 3.6 takes look at the UD analysis of various remaining cases, which might not have any standard grammatical description. As the aim of UD is to enable annotating every word in every utterance that might appear in any place where language is used, it needs to define some ways to deal with such cases.

The ambitious goal of this chapter is to enable the reader to analyse the dependency relations in arbitrarily complex sentences and thereby to perform tasks such as contributing to UD treebanks.

3.1 Syntactic relations in Universal Dependencies

Syntactic relation between words are the building blocks of dependency trees. The UD standard lists a set of relations in

`https://universaldependencies.org/u/dep/index.html`

Most of them are shown in shown in Table 3.1, just omitting some rare ones. The relations and their explanations are copied from the UD list. The third column shows the POS tags of the dependent-head pairs of words where the relation is typically used; this data is collected from

`en_pud-ud-test.conllu`, UD version 2.3

This is the Parallel UD treebank, which includes the same set of trees translated into several languages. This treebank will be repeatedly used in this book, as it is interesting for the interlingual perspective.

Usually at least one of the words can belong to several groups of POS tags:

- S, root words in sentences: VERB but also ADJ, NOUN, ADV, PROPN
- N, nominal classes: NOUN, PRON, PROPN, NUM
- C, (almost) any class

Some relations have subclasses, marked with a colon. The most important subclasses used for English are included in Table 3.1.

The treebank has dependency trees for 1000 sentences, in which there are a total of 21,183 words. The 20 most frequent syntactic relations are listed in Table 3.1, both by the relations alone and their combinations with dependent-head types.

Let us assume that we have an utterance tokenized, POS tagged, and morphologically analysed. Using the (simplified) CoNLL notation, we have then a list of word descriptions of the form

ID word lemma POS morpho

which we must complete by marking its head and the relation to the head:

ID word lemma POS morpho head relation

When we do this manually, a good way to proceed is **top down**:

relation	explanation	dependent-head	example
`acl`	clausal modifier of noun	S-N	*the moon as we* ***see*** *it*
`acl:relcl`	relative clause modifier	S-N	*the moon that we* ***see***
`advcl`	adverbial clause modifier	S-C	*I leave if she* ***goes***
`advmod`	adverbial modifier	ADV-C	*he sleeps* ***now***
`amod`	adjectival modifier	ADJ-N	***black*** *cat*
`appos`	appositional modifier	N-N	*Macron, the* ***president***
`aux`	auxiliary	AUX-S	***does*** *he sing*
`case`	case marking	ADP-N	***on*** *the moon*
`cc`	coordinating conjunction	CCONJ-C	***and*** *dogs*
`ccomp`	clausal complement	S-C	*I know that he* ***runs***
`compound`	compound	N-N	***data*** *science*
`conj`	conjunct	C-C	*cats and* ***dogs***
`cop`	copula	AUX-S	*he* ***is*** *old*
`csubj`	clausal subject	S-S	*that is* ***moves*** *is clear*
`dep`	unspecified dependency	C-C	(if nothing else works)
`det`	determiner	DET-N	***the*** *cat*
`expl`	expletive	PRON-S	***there*** *is hope*
`fixed`	fixed multiword expression	ADP-C	*because* ***of***
`flat`	flat multiword expression	PROPN-PROPN	*Adam* ***Smith***
`iobj`	indirect object	N-VERB	*she gave* ***us*** *a hint*
`mark`	marker	PART/SCONJ-S	***to*** *go*
`nmod`	nominal modifier	NOUN-NOUN	*man on the* ***moon***
`nmod:poss`	possessive modifier	N-NOUN	***my*** *cat*
`nsubj`	nominal subject	N-S	***John*** *walks*
`nsubj:pass`	nominal subject of passive	N-VERB	***John*** *was seen*
`nummod`	numeric modifier	NUM-N	***five*** *cats*
`obj`	object	N-VERB	*she sees* ***us***
`obl`	oblique nominal	N-S	*she comes with* ***us***
`parataxis`	parataxis	VERB-VERB	*I said:* ***come*** *here*
`punct`	punctuation	PUNCT-S	*I see*
`root`	root	S-	*John **walks*
`xcomp`	open clausal complement	S-S	*I want to* ***go***

Table 3.1: Syntactic relations used in UD standard 2, together with their typical uses. In the examples, the dependent is **boldfaced** and its head underlined. We have left out eight rare relations, but they are all explained in the text.

relation	occurrences
case	2499
punct	2451
det	2046
nsubj	1393
amod	1336
obl	1237
nmod	1076
obj	876
advmod	852
compound	810
conj	634
cc	574
mark	556
aux	410
nmod:poss	365
cop	316
advcl	293
aux:pass	274
xcomp	271
nummod	254

relation	occurrences
punct	2339
obl	1456
nsubj	1104
nmod:poss	966
obj	924
amod	909
advmod	872
conj	688
cc	585
nmod	393
flat:name	389
nsubj:cop	372
cop	365
aux	362
case	318
mark	313
nummod	311
advcl	283
det	245
acl:relcl	227

Table 3.2: Top-20 syntactic relations and their uses in English (left) and Finnish (right) Parallel UD (PUD) treebanks, which are translations of each other. The `root` relation appears once in every tree, hence 1000 times in the treebank. The table shows clearly two differences between the languages: Finnish uses case inflection instead of adposition, resulting in a low frequency of `case`, and has no articles, resulting in a low frequency of `det`.

1. mark the root of the tree (yes, the root is the top of the tree in this world!)
2. mark the immediate dependents of the root
3. mark the immediate dependents of these, and so on
4. until all words have been marked with exactly one head and relation

Let us now go through the relations in the order that they typically appear in the top down procedure. The discussion is in a way a condensed version of the UD annotation guidelines, which we have organized to support top-down analysis. We will moreover mention some problems in places where, from the interlingual perspective, the guidelines are not quite as universal as one could hope.

We will show several examples of each relation. In these examples, we will follow the convention to set the dependent word under discussion in **boldface**, and with its head underlined. Sometimes we will also mark the relation in square brackets after the example:

> ***she*** *has not* *walked* [`nsubj`]

3.2 The main clause and its parts

A **clause** is a phrase with a verb and its **arguments**, which are

- the **subject** of the verb
- the **complements** of the verb, which include one or more of
  - an **object**, also known as **direct object**, typically a noun phrase without a preposition
  - an **indirect object**, another noun phrase without a preposition (not very common)
  - **oblique objects**, noun phrases with prepositions

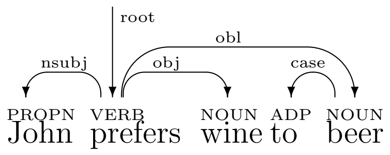
The **valency** of the verb determines which arguments it has. Thus for instance the verb *prefer* in the sentence

> *John prefers wine to beer*

has the arguments

- subject *John*
- direct object *wine*
- oblique object *to beer*

as shown in the full picture:



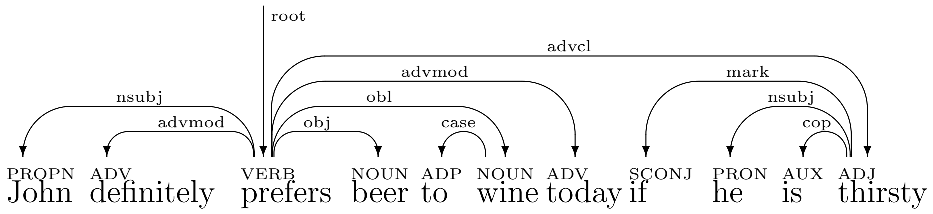
In addition to the more or less compulsory arguments, a clause can have **adjuncts**, which are optional modifiers. The valency neither requires nor restricts the number of adjuncts, so that they can be added or removed without affecting grammaticality:

John definitely prefers beer to wine today if he is thirsty

has the adjuncts

- *definitely*, a sentence adverbial
- *today*, an adverbial modifier
- *if he is thirsty*, a clausal modifier

Here is the full picture:


root
advcl
advmod
mark
nsubj
obl
nsubj
advmod
obj
case
cop
PROPN ADV VERB NOUN ADP NOUN ADV SCONJ PRON AUX ADJ
John definitely prefers beer to wine today if he is thirsty

Clausal modifiers are themselves clauses, which can have clausal modifiers, and so on. They are examples of **subordinate clauses**, which repeat the same structure with a verb and its arguments and adjuncts.

The concepts used above come from traditional grammar. They are also reflected in UD, which however makes some deviations from the tradition, as we will see below.

3.2.1 The root

The root of a clause is typically the main VERB in agreement of the main clause, ignoring auxiliary verbs (AUX):

*John **walks*** [`root`]

*John does not **walk*** [`root`]

*John has **walked*** [`root`]

If the verb is the **copula** (*be*), the root is its complement ADJ, NOUN, or ADV:

> *John is* ***old*** [`root`]
>
> *John is a* ***doctor*** [`root`]
>
> *John is* ***here*** [`root`]

If the utterance has no verb, the root can be of almost any category:

> *Probably the best* ***beer*** *in the world.* [`root`] (NOUN)
>
> ***Why*** *?* [`root`] (ADV)
>
> *She* ***will.*** [`root`] (AUX)

The choice of the root follows the general principle in UD:

> Heads are content words.

This is why other words become heads if the main verb is a copula. The same applies to **auxiliary verbs** such as *do* (when negating a verb) and *will* (marking the future tense). From the interlingual perspective, the principle makes sense, because function words are often not words in all languages: they can be inflectional features or just omitted, like for instance the copula in Russian or the future tense in French. In such cases, a function word as a head would make it impossible to build a valid dependency tree. Notice, however, the sentence *she will* (used e.g. as a confirmation) has an auxiliary as its head, since there is no main verb.

The root appears always in the **main clause**, as opposed to **subordinate clauses**. In subordinate clauses, the main verb (or similar) becomes the head of other words, and the dependent of the head of the dominating clause. We will return to subordinate clauses below.

For the reader familiar with symbolic logic, the root of a sentence is similar to a **predicate**, of which some other words are **arguments**. Thus we could translate Thus

> *John loves Mary* $\longrightarrow$ love(John,Mary)

In this tradition, it is natural that copulas are ignored, so that

> *John is old* $\longrightarrow$ old(John)

Auxiliaries and other modifiers may become operators outside the predicate, such as tense and negation:

> *John has not slept* $\longrightarrow$ not(Past(sleep(John)))

3.2.2 The subject

The **nominal subject** (`nsubj`) is a word in a **nominal category** (N in Table 3.1), that is, a NOUN, PRON, PROPN, or NUM, linked to the main verb (or other content word head) of the clause (root in the case of the main clause):

> ***John*** *walks* [`nsubj`]
>
> *the black* ***cat*** *walks* [`nsubj`]
>
> ***she*** *walks* [`nsubj`]

Notice that if the subject is a complex noun phrase such as *the black cat*, it is the head noun that is marked as `nsubj`. The dependents of nouns are discussed below, under `amod`, `det`, and `nmod`.

If the main content word is other than verb, the subject is in UD linked to it and not to the copula:

> ***John*** *is old* [`nsubj`]

This decision in UD differs from traditional grammar, where the subject is always the subject of a verb.

In English and some other languages, subjects of passive sentences are marked with the extended label `nsubj:pass`:

> *the black* ***cat*** *was seen* [`nsubj:pass`]

The purpose of this is to distinguish it from the "semantic subject" of the verb (the one who saw the cat). Semantic subjects of passives can be expressed as **agents**, but need not be expressed at all.

A clause can also have a **formal subject**, called **expletive** (`expl`) in UD. The expletives in English are *there* and *it*, classified as PRON:

> ***there*** *is an elephant in the room* [`expl`]
>
> ***it*** *is too cold in the room* [`expl`]

The head of the expletive *there* is the verb *be.* For *it*, the head is the same as if *it* were a normal *nsubj.*

But how do we distinguish the expletive *it* from a normal noun phrase? One possible test is to see if it can be an answer to a question:

> *What is too cold in the room?*

This is clearly not a quesion answered by ***it*** *is too cold in the room.*

In sentences with expletive *there* (see `expl` below), the subject is "the thing that exists" and comes usually after the verb:

> *there* <u>*is*</u> *an* ***elephant*** *in the room* [`nsubj`]

From the interlingual perspective, many languages may not have words for formal subjects at all. In particular, existentials (*there*) are expressed by a large number of different constructions. We will return to them in Chapter 5.

3.2.3 Complements and adjuncts

The **direct object** (`obj`), just like subject, is a word in a nominal category, linked to the main verb:

> *she* <u>*sees*</u> ***John*** [`obj`]
>
> *she* <u>*sees*</u> *the black* ***cat*** [`obj`]

The object is typically in the accusative case, if the language has such a case, but it can also be in some other case, such as dative in German:

> *Johann* <u>*folgt*</u> *diesen* ***Männern*** [`obj`] ("John follows these **men**")

The principle is:

- if the verb has just one non-prepositional completent, it is marked `obj`, whatever its case is.

The **indirect object** (`iobj`) is an additional complement to a verb that already has an object:

> *she* <u>*gave*</u> ***me*** *a hint* [`iobj`]

where *rose* is the `obj` object. The indirect object is typically in the dative case, if the language has one, but it can also be in some other case, even another accusative; notice that English makes no distinction between these cases.

Complements that have prepositions are marked as **oblique** (`obl`):

> *she* <u>*gave*</u> *a hint to* ***me*** [`obl`]

There is no distinction in UD between **complements** and **adjuncts**. Hence *me* is an oblique nominal attached to *walks* in

John <u>*walks*</u> *with* ***me*** [`obl`]

This is another place where UD departs from much of linguistic tradition, where both non-prepositional and prepositional complements can be attached to verbs in their valency patterns. The motivation is that it is sometimes difficult to decide whether a prepositional phrase is a complement or an adjunct. The price to pay is that it becomes more difficult to recognize corresponding arguments across languages, because what is `obj` or `iobj` in one language may come out as `obl` in another one.

Adverbial modifiers (`advmod`) are words of the ADV class linked to main verbs, but also to other words:

she <u>*walks*</u> ***today*** [`advmod`] (VERB)

genetically <u>*modified*</u> [`advmod`] (ADJ)

Also **interrogative adverbs** (*when, where, why*) are classified as ADV and linked to their heads as `advmod`:

why *does she* <u>*walk*</u> [`advmod`]

Negation words (*not* are also treated as `advmod`:

why *does she* ***not*** *walk* [`advmod`]

In this sentence, both *why* and *not* are adverbial modifiers of *walk*.

Notice that UD uses `advmod` only with words classified as ADV, and not with other adverbial phrases. Thus prepositional phrases (*in the city*) have the relation `obl` when linked to verbs, and `nmod` when linked to nouns (see the `nmod` section below).

3.2.4 Auxiliary verbs

The **copula** *be* is classified as AUX and linked to is complement (noun, adjective, adverb) with the `cop` relation:

she ***is*** <u>*here*</u> [`cop`]
she has ***been*** <u>*here*</u> [`cop`]

In UD, however, the verb *be* is sometimes treated as the main verb and classified as VERB. This is the case with existentials with the expletive *there* (`expl`, see below) *there* ***is*** *an elephant* [`root`]

and with clausal complements (`ccomp`, see below)

the reason ***is*** *that I am tired* [`root`]

AUX verbs other than the copula are linked to their main verbs with the `aux` relation:

he ***can*** <u>*sing*</u> [`aux`]

does *he* <u>*sing*</u> [`aux`]

English compound tenses can have several auxiliaries, each linked to the main verb independently:

would *he* ***have*** <u>*sung*</u> [`aux`]

From the interlingual perspective, a verb in a given function that is AUX in English migh not be AUX in other languages. For instance, the French equivalents of *can*, *pouvoir* and *savoir*, get the POS tag VERB and function as main verbs.

3.2.5 Punctuation

An utterance may or may not end with a punctuation mark (with the POS tag PUNCT). If it does, the mark becomes a dependent of the root with label `punct`:

Why does he <u>*walk*</u> *in the park* ***?*** [`punct`]

(Notice the boldface ***?***.) A punctuation mark can also attach to a part of the utterance:

he called me a ***"*** *bad* <u>*loser*</u> ***"*** [`punct`]

Both quotes are linked to the head of the phrase that they surround.

3.3 Dependents of nominals

3.3.1 Determiners

A noun can be **determined** (`det`) by a determiner (with POS tag DET). Also interrogative determiners (*which*) are analysed in this way.

> ***the*** *cat* [`det`]
>
> ***which*** *black cat* [`det`]

3.3.2 Modifiers

A noun can be modified in several ways. **Adjectival modification** (`amod`) is made by an adjective (ADV):

> ***black*** *cat* [`amod`]

The adjective can appear after the noun even in English,

> *movie* ***larger*** *than life* (`amod`)
>
> *nothing* ***wrong*** [`amod`]

Notice that in the latter example, taken from official UD documentation, the head *nothing* is not a noun in the standard sense. Other non-standard examples of adjectival modification are

> *million* ***dollar*** *loan* [`amod`]

where the modifier is a noun, adn

> ***more*** *questions* [`amod`]

where the modifier is a determiner rather than an adjective. Such corner cases are not always easy to decide.

One reason can be the tradition of English dictionaries to mark many different kinds of words as “adjectives”, on the basis that they are words prefixed to nouns: determiners and numerals are examples of this. The thinking behind this may be the substution test: since one can say each of

> *difficult questions, more questions, five questions*

one could conclude that *more* and *five* are subtitutible for *difficult* and hence adjectives. However, the proper application of the substitution test should look at *all* contexts, not only one. The following test would fail, with ungrammatical phrases marked with *:

*very difficult questions, *very more questions, *very five questions*

Nominal modifier (`nmod`) is typically a **prepositional phrase** (preposition + noun phrase) modifying a noun:

the house of the ***president*** [`nmod`]

a man on the ***moon*** [`nmod`]

A special case in English is **possessive modifier**, which can be a NOUN or a PRON:

the ***president****'s house* [`nmod:poss`]

my *house* [`nmod:poss`]

This is analogous to what happens in languages with rich case systems, where nominal modifiers often have no prepositions but cases such the Finnish local cases:

kylä *vuorella* [`nmod`] ("a village on a mountain")

Prepositions are a special case of **adpositions**, which can also be **postpositions** appearing after the noun. UD has the common POS tag ADP for both. They are linked to their heads by the `case` relation:

on *the moon* [`case`]

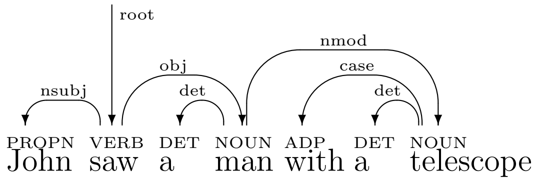
Postpositions are common in Finnish:

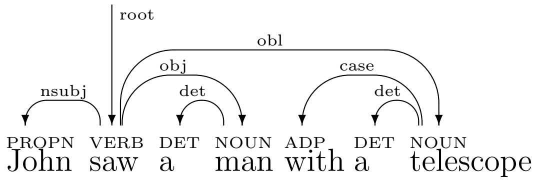
maidon ***kanssa*** [`case`] ("with milk")

The English possessive clitic *'s* has in UD the POS tag PART (particle), but is linked to its head as a `case`:

president ***'s*** [`case`]

A nominal modifier can often be analysed alternatively as an oblique object of the verb. This is known as the **PP attachement problem** (PP = Prepositional Phrase). Here is the standard example:


root
nmod
obj
case
nsubj
det
det
PROPN VERB DET NOUN ADP DET NOUN
John saw a man with a telescope


root
obl
obj
case
nsubj
det
det
PROPN VERB DET NOUN ADP DET NOUN
John saw a man with a telescope

Numeric modifiers (`nummod`) are syntactically similar to determiners but have their own relation in UD:

five *houses* [`nummod`]

40,000 *dollars* [`nummod`]

$ ***40,000*** [`nummod`]

Appositional modifiers (`appos`) are nominal modifiers that are (in English) attached to noun phrases, often but not always **proper names** (PROPN), and separated by punctuation:

Macron , the ***president*** [`appos`]

price : ***30*** *dollars* [`appos`]

They are in UD distinguished from **flat multiword expressions** (`flat`), which are used for titles but also parts of names:

president ***Macron*** [`flat`]

Emmanuel ***Macron*** [`flat`]

president ***Emmanuel Macron*** [`flat`] (two separate modifiers)

Notice the opposite direction of the head-dependent relation in `appos` and `flat` when assigning a title to a person.

Compound (`compound`) is a nominal modifier appearing in **compound nouns** in languages where they are written separately, as in English. This relation is also used for composite numeral expressions.

dependency *tree* [`compound`]

fifty *thousand* [`compound`]

3.4 Subordinate clauses and embedded verb phrases

Subordinate clauses are clauses that appear as arguments or modifiers of other phrases. Their internal structure is similar to main clauses, typically with a main verb (or some other word if the verb is a copula) and its dependents. The main word is not labelled as root, but as dependent of some word in the **dominating clause**.

3.4.1 Clausal complements

A typical **clausal complement** (`ccomp`) is a *that* clause that is a complement of a verb such as *say* or *believe*:

*I believe that the Earth **moves*** [`ccomp`]

The word *that* has the POS tag SCONJ (subordinating conjunction) and the relation `mark` to the main verb of the subordinate clause:

*I believe **that** the Earth moves* [`mark`]

In English and some other languages, it is common to leave out the mark:

*I believe the Earth **moves*** [`ccomp`]

Clausal complements are the analysis given to **indirect speech** ("X said that Y"), whereas **direct speech** ("X said: Y") is treated as `parataxis`:

*I said: the Earth **moves*** [`parataxis`]

*"The Earth **moves**", I said* [`parataxis`]

A tricky case of `ccomp` is clausal complements of copula+noun constructions:

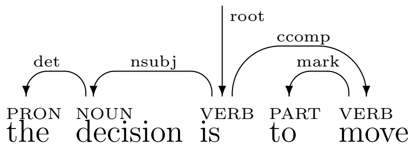
*my opinion is that the Earth **moves*** [`ccomp`]

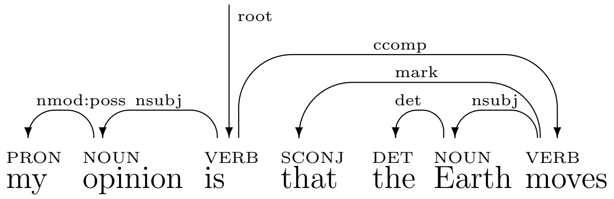
*the decision is to **move*** [`ccomp`]

In these cases, UD has decided to treat the copula as the main verb, "to preserve the integrity of clause boundaries and prevent one predicate to be assigned two subjects", admitting that "this is not an optimal solution"; see

`https://universaldependencies.org/u/dep/ccomp.html`

The mark (*that*, *to*), is related to the verb as `mark`, as shown by the full picture




root
ccomp
mark
nmod:poss
nsubj
det
nsubj
PRON NOUN VERB SCONJ DET NOUN VERB
my opinion is that the Earth moves

Another potentially tricky question is to distinguish `ccomp` from `xcomp`, **open clausal complement**. The difference is that the **logical subject** of the verb (the one who is thought to perform the action) in **xcomp** is determined by the dominating clause, as either its subject or object. In such constructions, the verb is typically in the infinitive form:

*I <u>want</u> to **move*** [`xcomp`]

*I <u>want</u> you to **move*** [`xcomp`]

*I <u>saw</u> the Earth **move*** [`xcomp`]

The infinite mark *to* is again related to the subordinated verb as `mark`:

*I want **to** <u>move</u>* [`mark`]

An `xcomp` can also be a noun or an adjective:

*I consider him **honest*** [`xcomp`]

*I consider him a **fool*** [`xcomp`]

The logic is that one can think of a missing copula: these sentences are treated in the same way as

I consider him to be ***honest*** [`xcomp`]

I consider him to be a ***fool*** [`xcomp`]

The infinitive mark here is of course linked to the adjective or noun:

I consider him ***to*** *be honest* [`mark`]

3.4.2 Clausal subjects

Subordinate clauses can also be on the subject position, **clausal subjects** (`csubj`):

what ***happens*** *does not concern me* [`csubj`]

to ***leave*** *now makes sense* [`csubj`]

A special case is, analogously to `nsubj`, passive subject position (`csubj:pass`):

what ***happens*** *has been decided before* [`csubj:pass`]

3.4.3 Clausal modifiers

Subordinate clauses in adverbial positions are called **adverbial clausal modifiers** (`advcl`):

I drank because I was ***thirsty*** [`advcl`]

if you ***leave*** *, I will resign* [`advcl`]

The SCONJ (such as *because, if, when, although*) leading the adverbial clause is a `mark`:

because *I was thirsty* [`mark`]

Clauses can also modify nouns, by the relation `acl` **clausal modifier of noun**:

the situation as I ***see*** *it* [`acl`]

the situation ***caused*** *by you* [`acl`]

the reason to ***leave*** [`acl`]

An important special case is **relative clause modifiers**, `acl:relcl`:

the girl who ***walks*** [`acl:relcl`]

the girl that I ***know*** [`acl:relcl`]

the girl I ***know*** [`acl:relcl`]

The relative pronoun (sometimes omitted in English) has the POS tag `PRON` and a morphological tag `PronType=Rel`. It is a normal dependent (such as subject or object) of the head of the relative clause:

the girl ***who*** *walks* [`nsubj`]

the girl ***that*** *I know* [`obj`]

3.5 Coordination structures

Coordination means joining two or more phrases of the same category with a **conjunction**, such as *and*, *or*. It can be performed with almost any types of phrases: sentences, verbs, nouns, adjectives, adverbials, and so on. The members of a conjunction are called **conjuncts**. The UD standard uses the relation `conj` that treats the first conjunct as head and the other members its dependents:

John walks and Mary ***runs*** [`conj`]

he is hungry and ***tired*** [`conj`]

I want to drink milk , ***coffee*** *or* ***beer*** [`conj`] (two dependents)

The conjunction word itself has POS tag `CCONJ` (**coordinating conjunction**) and is linked to the conjunct that it precedes with the `cc` relation (likewise called **coordinating conjunction**):

hungry ***and*** *tired* [`cc`]

Conjunctions such as *both-and* and *either-or* have **preconjunctions**, which are dependents of the first conjunct by the `cc:preconj` relation:

both *hungry and tired* [`cc:preconj`]

The commas that are typically used to separate conjuncts if there are more than two of them have their usual POS tag `PUNCT` and the relation `punct` (**punctuation**) whose head is the conjunct that the comma precedes:

milk ***,*** *coffee or beer* [`punct`]

The dependency analysis of coordinating structures is not entirely natural, and UD has chosen a convention different from many other approaches. The reason is that the relation between conjunct is not really dependency, but a more symmetric relation. We will return to this problem later when we look at coordination from the phrase structure point of view.

3.6 Remaining relations

We have now discussed the most important grammatical relations. One can get a long way by just using them — both in English and in other languages. The remaining ones are:

- `clf`, **classifier**, is a word in Asian languages such as Chinese and Thai used when combining nouns with numerals and determiners. Its head in UD is not the noun but the determiner: *wu* ***zhi*** *mao* "five cats" Chinese
- `dep`, **unspecified dependency**, used as fall-back when no other relation seems to apply, e.g. in plainly ungrammatical utterances
- `discourse`, **discourse element**, used for interjections and smileys: ***Hey****, come here!*
- `dislocated`, **dislocated element**, an element before or after the sentence structure, typically separanted by comma: *He is not nice,* ***John***
- `fixed`, **fixed multiword expressions**, in particular function words and adverbials: *because* ***of***
- `list`, **list**, between related items where there is no conjunction: *name John,* ***title*** *doctor*
- `goeswith`, **goes with**, for parts words that appear separated in a text although they should normally be written together: *with* ***out*** *doubt*
- `orphan`, **orphan**, used e.g. to relate the object of an omitted verb to the subject: *John loves Mary and Bill* ***Jane***
- `reparandum`, **overridden disfluency**, used in self-correcting utterances that would otherwise become ungrammatical: *I have five — eh,* ***six*** *cats*
- `vocative`, **vocative**, addressing a hearer or hearers: *Come here,* ***John****!*

Some of these relations are clearly existing grammatical relations: classifier, discourse element, dislocated, orphan, vocative. Some are there to enable the analysis of even "ungrammatical" utterances: goes with, reparandum.

The `dep` relation is used to guarantee that every word gets a label, even if the human annotator or the parsing algorithm does not know what the label should be.

Chapter 4

Grammatical analysis: phrase structure

In this chapter, we continue with syntactic analysis but now in terms of phrases rather than dependencies. In fact, when explaining dependency analysis in UD, we often have to refer to phrases and not just words. For example, a clausal complement (`ccomp`, Section 3.4) is meant to be the whole subordinate clause and not just its main verb.

Like in the previous chapter, we will proceed by a top-down analysis from sentences to their parts. We will loosely follow the phrase structure used in the **GF Resource Grammar Library** (RGL),

`http://www.grammaticalframework.org/lib/doc/synopsis/`

The structures of RGL are designed to work for multiple languages, just like UD. The phrase structure in this chapter is designed to be in close correspondence with both RGL and UD, with some bias to UD when necessary. In Chapter 6.4, we will complete the description to an exact match with the RGL.

We could still work under the assumption that the analysis is carried out manually, but will complete this with a rule format that supports a **parsing algorithm** that can partly automate the analysis. This format is called **Dependency Backus Naur Form** (DBNF), which extends the standard **Backus Naur Form** (BNF) with dependencies. Our DBNF rules are intended for analysis, without trying to define what is grammatical and what is not: in fact, they are vastly overgenerating and would allow lots of unwanted analyses if applied mechanically. Nevertheless, they are more precise than the completely informal guidelines used for dependency annotation in the UD documentation and in the previous chapter.

The structure of this chapter is as follows:

- Section 4.1 gives a list of the main categories of words and phrases as implemented in the GF Resource Grammar Library.
- Section 4.2 specifies the DBNF format of phrase structure rules with dependences, as well as phrase structure trees.
- Section 4.3 goes through a set of DBNF rules for English.
- Section 4.4 discusses briefly the generalizability of the rules to other languages.English.
- Section 4.5 shows how the grammar rules used in this chapter can be used for automated analysis by using the program `gfud`.

4.1 Categories

Table 4.1 lists some categories from the GF Resource Grammar Library (RGL). It is divided into two parts:

- **lexical categories**, which appear in the **leaves** of phrase structure trees (in nodes that have no branches),
- **phrasal categories**, which appear higher up in trees and combine phrases into larger structures.

The lexical categories are analogous to the universal part of speech tags used in UD. The phrasal categories can be thought of as collecting groups of words, each group consisting of a head and its dependents. Both lexical and phrasal categories are *universal* in the sense that they are meant to apply to many languages.

We will postpone the interlingual perspective to later chapters and concentrate on English in this chapter. However, it is useful to know that the analysis we give is — just like the UD analysis of the previous chapter — applicable to a wide variety of other languages as well, if we abstract away from morphology, the number of words, and word order.

4.2 The DBNF rule format

We will use a traditional format for phrase structure rules, known as the **Backus-Naur Form** (**BNF**). An example is

category	explanation	example	UD
`A`	adjective	*warm*	ADJ
`Adv`	adverb	*now*	ADV
`Card`	cardinal number	*seven*	NUM
`Conj`	conjunction	*and*	CCONJ
`Det`	determiner	*the*	DET
`Digits`	numeral in digits	*1,000,002*	NUM
`IAdv`	interrogative adverb	*why*	ADV
`IDet`	interrogative determiner	*which*	PRON
`IP`	interrogative pronoun	*who*	PRON
`N`	common noun	*house*	NOUN
`PN`	proper name	*Paris*	PROPN
`Predet`	predeterminer	*only*	DET
`Prep`	preposition	*in*	ADP
`Pron`	personal pronoun	*she*	PRON
`Punct`	punctuation mark	*!*	PUNCT
`RP`	relative pronoun	*which*	PRON
`Subj`	subjunction	*if*	SCONJ
`V`	verb	*sleep*	VERB

category	explanation	example
`AP`	adjectival phrase	*very warm*
`CN`	common noun phrase	*red house*
`Comp`	complement of copula, such as AP	*very warm*
`NP`	noun phrase (subject or object)	*the red house*
`Num`	numeral	*seven hundred*
`QS`	question	*where did she live*
`RS`	relative clause	*in which she lived*
`S`	declarative sentence	*she lived here*
`SC`	embedded sentence or question	*that it rains*
`Utt`	sentence, question, word...	*It is OK.*
`VP`	verb phrase	*is very warm*
`VPSlash`	verb phrase missing complement	*look at*

Table 4.1: Some lexical and phrasal categories in GF Resource Grammar Library (RGL), as usen in this chapter. The UD column in lexical categories gives the closest corresponding Universal POS tags.

```
S    ::= NP VP
NP   ::= Pron
VP   ::= Cop A
Pron ::= "she"
Cop  ::= "is"
A    ::= "old"
```

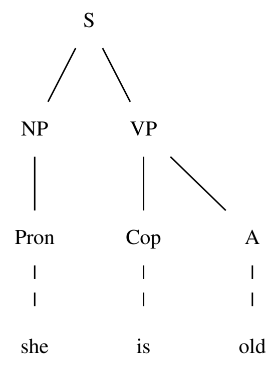
The category left of the symbol `::=` is called the **left-hand side** (LHS), and the part after is the **right-hand side** (RHS). The categories appearing in these rules are called **non-terminals**, and quoted strings are called **terminals**.

> A phrase of the LHS category can be built by combining the phrases of the categories and terminals on the RHS.

For example, the first rule says,

> A sentence (S) can be formed from a noun phrase (NP) followed by a verb phrase (VP).

With BNF rules, we can build **parse trees**, which encode each application of a rule by a **subtree** where the uppermost **node** is the LHS category of the rule that was applied. Hence, with the rules given above, we can form the tree



Another way to display the tree is as a **bracketed string**, which uses parentheses and category symbols to show the structure of the expression:

> S (NP (Pron *she*) (VP (Cop *is*) (A *old*)))

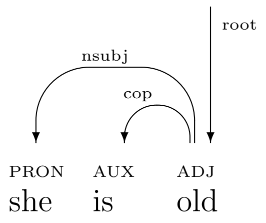
The strings corresponding to subtrees are called **constituents**. Constituents can have their own constituents, so that for instance both *is old* and *old* are constituents in this example.

Dependency BNF (**DBNF**) completes the BNF notation with dependency labels and (optionally) weights of each rule. The notation is explained in Figure 4.2.

DBNF offers a quick way to relate phrase structure with dependency analysis. The labels mark which subtrees contain the words that get each of the labels. The `#pos` rules give their POS tags. Using the DBNF rules

```
S    ::= NP VP # nsubj head
VP   ::= Cop A # cop head
#pos PRON Pron
#pos AUX Cop
#pos ADJ A
```

we get the following dependency tree for *she is old*:



DBNF is a variant of **context-free grammars**, which have well-known parsing algorithms. The rules that we will show in this chapter are from an English grammar defined in

```
https://github.com/GrammaticalFramework/gf-ud/blob/master/
grammars/English.dbnf
```

The same Git repository contains a parser that can use the grammar to produce both phrase structure and dependency trees. The last explanation in Figure 4.2 means that the lexicon is open: the parser can accept input that is tagged by POS tags. Hence for instance the following input would parse:

```
I:<PRON> am happy:<ADJ>
```

The DBNF rule format:

- **Syntax rule**: *cat* `::=` *cats* `#` *labels* `#` *weight*
  Example: `S ::= NP aux?  VP # nsubj aux head # 0.02`
- **POS tag rule**: `#pos` *postag cats*
  Example: `#pos VERB V V2`
- **Token rule**: `#token` *cat words*
  Example: `#token Cop am is are was were`

Explanations:

- Dependency labels are separated from the BNF rule by the `#` symbol; the default is `head`.
- After another `#`, a **weight** between 0 and 1 can be given; the default is 0.5.
- The number of labels must be the same as the number of nonterminals on the right hand side.
- Exactly one label must be `head`.
- If there is exactly one nonterminal, the label `head` is assumed by default.
- Nonterminals with suffix `?` are optional; the nonterminal corresponding to `head` may not be optional.
- Lexical categories with POS tag `P` are listed in `#pos P` rules.
- Terminals of a category `C` can be listed in `#token C` rules.
- In addition to tokens in `#token C`, tokens of a category `C` include **ad hoc tokens** given in the input in the form `WORD:<P>`, where `C` is listed in `#pos P`. Their weight is 0.2.

Figure 4.1: The Dependency BNF (DBNC) notation.

The grammar shown in this chapter is vastly overgenerating, because we don't try to model morphology and agreement. Overgeneration as such is fine when the task is parsing and not generation. However, it can lead to much more ambiguity than a strict grammar would allow. The system of **weights** can be used to mitigate this: the parse trees are ordered by descending weight, where the weight of a tree is the product of the weights of the rules applied when building the tree. The default weight 0.5 implies that smaller trees are preferred to larger ones. The lower weight of ad hoc tokens (given as `word:<POS>`), 0.2, means that ad hoc tokens are overridden by explicitly given tokens.

The weight mechanism is a quick and dirty way to disambiguate and does not always yield the right results. Technically it means that DBNF supports **Probabilistic Contex-Free Grammars** (PCFG), where context-freeness means that the weight of each rule is independent of the other rules. The probability of a tree is the probability of the rule forming the tree multiplied by the probabilities of subtrees:

$$P(R\,t_1 \ldots t_n) = P(R) \times P(t_1) \times \ldots \times P(t_n)$$

This is a simplifying assumption, since in reality combinations of rules should have special weights. In particular, the choice of words, not just their categories, can affect what rules are probable for combining them. Thus the sentence

John drinks beer

should be more probable than

beer drinks John

even though the two sentences are built with exactly the same rules and their context-free probabilities are therefore equal.

4.3 English phrase structure in DBNF

We will go through similar structures and examples as in Chapter 3, but now in terms of phrase structure analysis. The DBNF rules that we show can automatically convert the phrase structure analysis to UD dependency analysis.

4.3.1 Utterances

From the top-down perspective, the starting point of analysis is an **utterance** (Utt). Utterances can be sentences, questions, imperatives, noun phrases, adverbs, and so on, and they can optionally have punctuation.

```
Utt ::= Utt Punct # head punct
```
Utt (Utt *Why*) (Punct *?*)

```
Utt ::= S
```
Utt (S *John walks*)

```
Utt ::= QS
```
Utt (QS *why does she walk*)

```
Utt ::= Imp
```
Utt (Imp *wash your hands*)

```
Utt ::= NP
```
Utt (NP *probably the best beer in the world*)

```
Utt ::= Adv
```
Utt (Adv *certainly*)

Notice that the bracketed tree notation here does not show the full analysis, but just the division into immediate parts mentioned on the RHS of the BNF rule that is exemplified. This is analogous to only showing one dependent-head relation in the examples of Chapter 3.

4.3.2 Sentences

Following a common linguistic tradition, we will use the term **sentence** (S) for clauses. (The full GF-RGL will, however, make a distinction, to which we return later.) The most well-known rule constructs a sentence from a **noun phrase** (NP) and a **verb phrase** (VP):

```
S ::= NP VP # nsubj head
```
S (NP *the black cat*) (VP *sees us now*)

Between the NP and the VP, there can be auxiliaries and a negation:

```
S ::= NP do neg VP # nsubj aux advmod head
```
S (NP *the black cat*) (do *does*) (neg *not*) (VP *see us*)

`S ::= NP have neg?  VP # nsubj aux advmod head`
S (NP *the black cat*) (have *has*) (neg *not*) (VP *seen us*)

`S ::= NP aux neg?  have?  VP # nsubj aux advmod aux head`
S (NP *the black cat*) (aux *would*) (neg *not*) (have *have*) (VP *seen us*)

Notice the special non-terminals `do`, `have`, `neg`, and `aux`. They are defined as lists of special tokens:

```
#token do do does did done
#token have have has had
#token neg not
#token aux will would shall should can must
```

We write their categories with small initials to distinguish them from the RGL categories proper in Table 4.1. As we shall see later, they can and should be treated in a more abstract way in the interlingual perspective.

Sentences can also be formed with a copula (cop) and its complement (Comp).

`S ::= NP cop neg?  Comp # nsubj cop advmod head`
S (NP *John*) (cop *is*) (neg *not*) (Comp *happy*)

`S ::= NP have neg?  cop Comp # nsubj aux advmod cop head`
S (NP *John*) (have *has*) (neg *not*) (cop *been*) (Comp *happy*)

`S ::= NP aux neg?  have?  cop Comp # nsubj aux advmod aux cop head`
S (NP *John*) (aux *would*) (neg *not*) (have *have*) (cop *been*) (Comp *happy*)

The copula token is defined by

```
#token cop am are is was were being be
```

The NP, VP, and Comp categories will be explained below.

The above rules list different combinations of the verb and `aux`, `do`, `have`, and `neg`. Also a **sentence adverbial** (AdV) can be added, typically before the negation and the main verb:

`S ::= NP have AdV neg?  VP # nsubj aux advmod head`
S (NP *the black cat*) (have *has*) (AdV *definitely*) (VP *seen us*)

We will need many such combinations in the clause-forming rules that follow, but usually leave them implicit: full details can be found in English.dbnf.

Passive sentences are formed with a copula (and all its combinations with negation and auxiliaries) from passive verb phrases:

```
S ::= NP cop VP_pass # nsubj:pass aux:pass head
```
S (NP *John*) (cop *is*) (VP_pass *invited*)

The liberal rule for `VP_pass` is that it is just the same as VP: since we do not specify the morphological forms of verbs, then for instancs *invites* and *invited* have the same possible combinations. In English.dbnf, a bit more restrictive definition is given to decrease overgeneration.

4.3.3 Verb phrases

A **verb phrase** (VP) consists of a verb (V) with its **complements**. The **arguments** of a verb also include the subject, which is assigned on the sentence level as shown above. A complement is a direct or indirect noun phrase (NP) object or an oblique NP, that is, and NP with a preposition:

```
VP ::= V
```
VP (V *walk*)

```
VP ::= V NP # head obj
```
VP (V *see*) (NP *the black cat*)

```
VP ::= V NP NP # head iobj obj
```
VP (V *give*) (NP *me*) (NP *a hint*)

```
VP ::= VP NP_obl # head obl
```
VP (VP (V *give*) (NP *a hint*)) (NP_obl *to me*)

Notice that the last rule allows the addition of arbitrarily many oblique NPs to a VP.

The category `NP_obl` is an example of a **help category** that we introduce in DBNF to force the same dependency relations as in UD. It is produced by adding a preposition (Prep) to an NP:

```
NP_obl ::= Prep NP # case head
```
NP_obl (Prep *with*) (NP *me*)

We could have been able to parse the same sentences by writing directly

```
VP ::= VP Prep NP # head case obl
```

However, the resulting structure would attach the preposition to the VP and not to the NP. We mark help categories with underscores and suffixes; the category symbol before the underscore is a genuine RGL category, and the only category we will need in GF.

In addition to noun phrases, a complement can also be a sentence (S_that), a question (QS), an adjectival phrase (AP), or a verb prase:

```
VP ::= V S_that # head ccomp
```
VP (V *believe*) (S_that *the Earth moves*)

```
VP ::= V QS # head ccomp
```
VP (V *wonder*) (QS *who sees us*)

```
VP ::= V AP # head xcomp
```
VP (V *become*) (AP *old*)

```
VP ::= VP VP_to # head xcomp
```
VP (V *want*) (VP *to leave*)

The help categories `S_that` and `VP_to` introduce a mark, optional for `S_that`:

```
S_that ::= that? S    # mark head
VP_to  ::= neg? to VP # advmod mark head
```

The above VP rules are overgenerating as they do not take into accound **verb valencies**: what arguments are possible for what verbs. They could be made more precise by introducing some valency distinctions, for instance, V2 for 2-place verbs (taking one NP complement) and VS for sentence-complement verbs:

```
VP ::= V2 NP # head obj

VP ::= VS S_that # head ccomp
```

However, this would not block unwanted verb-complement combinations altogether, because of the open lexicon principle of DBNF and the POS rule

```
#pos VERB V V2 VS...
```

This POS rule reflects the fact that UD makes no distinctions between verb valencies. In the English.dbnf grammar that we use, some subcategories of V present, to give priority to analyses where verbs are actually declared in these subcategories. But this is just a soft distinction (i.e. a difference in weight), because of the open lexicon principle.

4.3.4 Complements of the copula

Complements come from different categories. The examples below show their use in sentences.

```
Comp ::= AP
```
S (NP *we*) (cop *are*) (Comp (AP *happy*))

```
Comp ::= NP
```
S (NP *we*) (cop *are*) (Comp (NP *the champions*))

```
Comp ::= NP_obl
```
S (NP *we*) (cop *are*) (Comp (NP_obl *in trouble*))

```
Comp ::= Adv
```
S (NP *we*) (cop *are*) (Comp (AP *here*))

Clausal complements are treated differently, to match the UD conventions stated in Section 3.4:

```
S ::= NP be_V SC # nsubj head ccomp
```
S (NP *the fact*) (be_V *is*) (SC *that the Earth moves*)

Notice that `be_V` is different from `cop`, as it appears as the main verb. The category SC, of **embedded clauses**, can be formed in different ways:

```
SC ::= S_that
```
S (NP *the fact*) (be_V *is*) (SC (S *that the Earth moves*))

```
SC ::= QS
```
S (NP *the question*) (be_V *is*) (SC (S *what moves*))

```
SC ::= VP_to
```
S (NP *the decision*) (be_V *is*) (SC (S *to move*))

SC can of course also appear in the subject position:

```
S ::= SC VP # csubj head
```
S (SC *that the Earth moves*) (VP *puzzles me*)

```
S ::= SC cop Comp # csubj cop head
```
S (SC *that the Earth moves*) (cop *is not*) (Comp *difficult to understand*)

4.3.5 Questions, relatives, and imperatives

Questions (QS) are similar to sentences (S), except for some word order differences and, most importantly, that they allow **interrogative phrases** (IP) as subjects and objects:

`QS ::= do NP VP # aux nsubj head`
QS (do *does*) (NP *John*) (VP *walk*)

`QS ::= cop NP Comp # cop nsubj head`
QS (cop *is*) (NP *John*) (Comp *happy*)

`QS ::= cop NP VP_pass # aux:pass nsubj:pass head`
QS (cop *is*) (NP *John*) (VP_pass *invited*)

`QS ::= IP VP # nsubj head`
QS (IP *who*) (VP *walks*)

`QS ::= IP cop Comp # nsubj cop head`
QS (IP *who*) (cop *is*) (Comp *there*)

`QS ::= IP do NP VPSlash # obj aux nsubj head`
QS (IP *whom*) (do *does*) (NP *the cat*) (VPSlash *see today*)

`QS ::= IP cop VP_pass # nsubj:pass aux:pass head`
QS (IP *who*) (cop *is*) (VP *seen today*)

A question can also be formed with an **interrogative adverbial** (IAdv):

`QS ::= IAdv do NP VP # advmod aux nsubj head`
QS (IAdv *why*) (do *does*) (NP *John*) (VP *walk*)

`QS ::= IAdv cop NP Comp # advmod cop nsubj head`
QS (IAdv *why*) (cop *is*) (NP *John*) (Comp *happy*)

`QS ::= IAdv cop NP VP_pass # advmod aux:pass nsubj:pass head`
QS (IAdv *why*) (cop *is*) (NP *John*) (VP_pass *invited*)

The category VPSlash, **verb phrase missing a complement**, could be identified with VP, as long as we don't care about verb valencies. However, in the RGL it plays an important role, because the proper generation of sentences including it need information that is not present in a VP — most importantly, the case of the missing complement. Thus for instance in German, we distinguish between *wen sieht er* "whom does he see", accusative

wem folgt er "whom does he follow", dative

The category itself comes from a linguistic tradition where it is called VP/NP ("VP slash NP") and was originally introduced to enable phrase structure grammars to model **wh movement**, that is, sentences where an object "moves" from its "original" place to the front when it is a "wh phrase" such as an interrogative or relative pronoun.

Relative clauses (RS) are indeed similar to questions with IP replaced by RP (**relative pronoun**). In English, the RP can also be omitted when it is the object:

```
RS ::= RP VP # nsubj head
```
RS (RP *who*) (VP *walks*)

```
RS ::= RP cop Comp # nsubj cop head
```
RS (RP *who*) (cop *is*) (Comp *there*)

```
RS ::= RP NP VPSlash # obj nsubj head
```
RS (RP *whom*) (NP *the cat*) (VPSlash *sees today*)

```
RS ::= NP VPSlash # nsubj head
```
RS (NP *the cat*) (VPSlash *sees today*)

```
RS ::= RP cop VP_pass # nsubj:pass aux:pass head
```
RS (RP *who*) (cop *is*) (VP *seen today*)

Notice the differences in word order and the use of *do*, when compared to questions. In these respects, relative clauses are in fact closer to **indirect questions**, which are used as embedded clauses:

S (NP *I*) (VP (V *wonder*) (QS *whom the cat sees*))

The overgenerating English.dbnf grammar includes indirect questions in QS. In the precise RGL grammar, they are also included in QS, but overgeneration is prevented by techniques that are stronger than context-free rules.

Imperatives are yet another type of clauses with a verb, but without a subject:

```
Imp ::= do?  neg?  VP # aux advmod head
```
Imp (do *do*) (neg *n't*) (VP *move*)

```
Imp ::= do?  neg?  cop Comp # aux advmod cop head
```
Imp (do *do*) (neg *n't*) (cop *be*) (Comp *silly*)

4.3.6 Noun phrases, adjectives, and adverbials

The most common ways of building **noun phrases** are from proper names (PN), pronouns (Pron), and common nouns (CN):

`NP ::= PN`
NP (PN *San Remo*)

`NP ::= Pron`
NP (PN *we*)

`NP ::= Det?  CN # det head`
NP (Det *the*) (CN *American imperialism*)

The difference between NP and CN is that an NP can occur as subject or object, wereas a CN typically requires a determiner (Det). However, a bare CN can also work as an NP, for instance, if it is in plural form (***tigers*** *are dangerous*) or if it a mass term (***French wine*** *is good*).

From the other NP-forming rules, let us list those that introduce new grammatical relations:

`NP ::= Predet NP # det:predet head`
NP (Predet *only*) (NP *this country*)

`NP ::= NP_poss CN # nmod:poss head`
NP (NP_poss *your*) (CN *country*)

`NP ::= Num NP # nummod head`
NP (Num *fifty thousand*) (CN *people*)

`NP ::= CN PN # head flat`
NP (CN *president*) (PN *Macron*)

`NP ::= NP SC # head acl`
NP (NP *the reason*) (SC *why we are here*)

`NP ::= NP comma?  S_sub # head punct acl`
NP (NP *our country*) (S_sub *as we know it*)

`NP ::= NP comma?  RS # head punct acl:relcl`
NP (NP *the country*) (RS *that we love*)

`NP ::= NP comma?  VP_pass # head punct acl`
NP (NP *the palace*) (VP_pass *built for the president*)

`NP ::= Symb Num # head nummod`
NP (Symb *$*) (Num *15 million*)

Common nouns (CN) are ultimately built from single nouns (N) and optionally modified by adjectival phrases (AP):

```
CN ::= N
```
CN (N *cat*)

```
CN ::= AP CN # amod head
```
CN (AP *big*) (CN *black cat*)

```
CN ::= CN comma?  AP # head punct amod
```
CN (CN *movie*) (AP *larger than life*)

Common nouns can also be modified by relative clauses, but this case is covered by relative modification of NPs. There is a semantic distinction between these two, which the RGL makes, but since UD does not, we leave it out for the moment.

Compound nouns can be built as just combinations of two nouns, and proper names (PN) can be combined in the same way:

N ::= N N # compound head
N (N *dependency*) (N *tree*)

PN ::= PN PN # head flat
PN (PN *San*) (PN *Remo*)

If more than two nouns are combined this way, a large number of parse trees may result (known as the Catalan number). English.dbnf mitigates this by introducing special categories N_one and PN_one, which imposes a linearly progressive order. But this may be unwanted, since one might want to give different structures to compounds on semantic grounds.

Adjectival phrases (AP) are ultimately built from adjectives (A), possibly modified by **adadjectives** (AdA) and oblique post-modifiers, in particular for comparative adjectives:

```
AP ::= A
```
AP (A *curious*)

```
AP ::= AdA AP # advmod head
```
AP (AdA *too*) (AP *curious*)

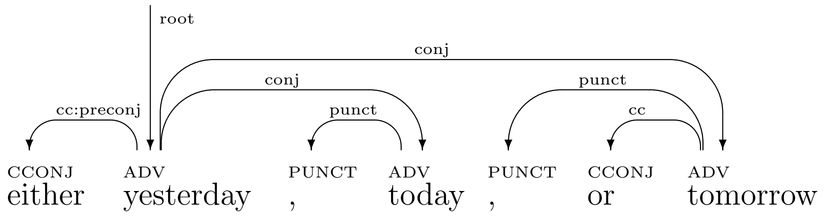
```
AP ::= AP NP_obl # head nmod
```
AP (AP *larger*) (NP_obl *than life*)

4.3.7 Coordination

Coordination is possible for many phrasal categories — S, NP, Adv, VP, RS, and so on. A possible general form of rules for a category X is

```
X ::= Conj_pre?  X_list X_conj # cc:preconj head conj
X_list ::= X
X_list ::= X_list X_comma # head conj
X_conj ::= comma?  Conj X # punct cc head
X_comma ::= comma X # punct head
```

This looks a bit complicated with as many as three help categories, but it gives the exact structure specified in UD annotation quidelines:



4.4 Phrase structure in other languages

Context-free phrase structure parsing can work reasonably well in languages that share two characteristics with English:

- rigid word order,
- simple morphology.

Hence it may also work with Scandinavian and East-Asian languages and with modern Romance languages. Morphology, in particular, is not so crucial when analysing language if it follows from agreement rules and does not carry much information of its own. Finnish and Latin are examples of languages at the opposite end: almost free word order, complex morphology that carries a lot of information. For example, the subject and the object of a verb is often recognizable only from the case of the noun phrase, as the order may vary because of emphasis or topicalization.

The simplest way to see the applicability is to take the grammar English.dbnf and modify it for another language. Some rules may stay exactly the same — for instance, since English has adjectival modifiers both before and after nouns,

```
CN ::= AP CN # amod head

CN ::= CN AP # head amod
```

these rules work for French, Italian, and Spanish as well, even though in those languages post-modifiers are much more common.

In some other cases, simply changing the word order, or adding more alternatives, can work: adding

```
NP_obl ::= NP Prep # head case
```

would allow postpositions in languages like Finnish. If more accuracy is required, a special category `Pre_post` can be introduced; however, Finnish often allows the use of one and the same adposition as both pre- and post-positions.

A more fundamental restriction of BNF rules is that they do not permit **discontinuous constituents**. Verb phrases (VP) is a typical example in languages that form questions by **inversion** of the subject and the verb. For instance, in Swedish, the VP in a declarative sentence is "split into two parts" when a question is formed:

S (NP *du*) (VP (V *bor*) (Adv *här*)) "you live here"

QS (V *bor*) (NP *du*) (Adv *här*) "do you live here"

Hence we cannot maintain the same phrase structure in these two languages.

Assigning the same structure to different languages is the leading principle in UD as well as in GF. In GF, it can be followed while at the same time maintaining the integrity of phrases. For instance the VP category can be defined as a discontinuous constituent, which allows us to use VP uniformly in the abstract syntax of many different languages.

4.5 Parsing with DBNF

A DBNF grammar can be used in the `gfud` program for parsing text and visualizing the results. It reads and writes standard input-output (stdio), which means that one can give it input and read its output in a Unix shell. It needs the grammar file and the start category as arguments: Here is a minimal example:

```
$ echo "John loves Mary" | gfud dbnf English.dbnf Utt

# text = John loves Mary
# analyses = 1
# parsetree = (Utt (S (NP (PN John)) (VP (V2 loves) (NP (PN Mary)))))
1    John    _    PROPN    _    _    2    nsubj    _    _
2    loves   _    VERB     _    _    0    root    _    _
3    Mary    _    PROPN    _    _    2    obj    _    _
```

The result is valid CoNLL output, where comment lines are prefixed with `#` and have standard meanings:

- The `# analyses` line tells how many parse trees were built; 1 means that the sentence was unambiguous, 0 that it was not parsed at all.
- The `# parsetree` line shows the phrase structure tree as a bracketed string.

As usual in Unix, the `echo` command can be used for parsing one example, whereas `cat` can read files. The output can be sent to a file with the > operator: Hence

```
cat examples.txt | gfud dbnf English.dbnf Utt >examples.conllu
```

creates a CoNLL file by parsing a text file. The text file is read line by line, and each line is expected to be tokenized. As specified in Section 4.2, a token given with a POS tag (`word:<POS>`) is parsed as a word of any category that matches the pos tag.

If the parser feels slow, the usual reason is a high number or parses. This can be observed from the `# analyses` line. It is possible to cut the number of parses by the `-cut` flag and a maximal number:

```
$ gfud dbnf English.dbnf Utt -cut=12
```

Only 12 trees are constructed in this case, but they are still ranked according to their weights, and the best one is shown. If more trees are wanted, the `-show` flag can be used (with or without the `-cut` flag):

```
$ gfud dbnf English.dbnf Utt -show=2
```

The `gfud` program also has visualization tools, for both CoNLL and bracketed trees:

```
$ cat examples.conllu | gfud conll2latex >examples.tex
```

writes a LaTeX file, which can be converted to pdf. This conversion is done automatically by

```
$ cat examples.conllu | gfud conll2pdf
```

which creates a temporary latex file, processes it, and opens it with the `open` command (available in MacOS with that name, but in other systems one may need to create an alias to some other pdf reader). A similar service is available for parse trees of the bracketed form:

```
$ cat examples.conllu | gfud parse2latex examples
$ cat examples.conllu | gfud parse2pdf
```

The first command writes the latex file `example.tex`. It needs a file name, because it also writes a number of other files, one for each tree to be shown, and these files are named using the same file name as a part. The pdf printing command creates temporary files with a default name.

The parse trees are expected to be each on its own line, prefixed by `# parsetree =`. These lines can be extracted from the CoNLL output of `gfud`. One can also skip the dependency trees and just output the parse trees:

```
$ gfud dbnf English.dbnf Utt -onlyparsetrees
```

The processing of temporary files when visualizing parse trees requires the GraphViz program called by the command `dot`. This makes it slow to visualize a large number of trees. Dependency tree visualization generates native LaTeX and is therefore much faster.

The DBNF parser implements a simple form of **robustness** by using **chunking**. The chunking algorithm identifies a sequence of subtrees that together cover all words of the input (in a left-to-right longest-match way). These subtrees are connected together into a category called `Chunks`, where the last chunk is treated as the head and the others are linked to it with the `dep` label. Unknown words are categorized as `Str`, with POS tag `X` In this way, the parser manages to give trees to all input, which is the expected behaviour in the UD tradition. Figure 4.5 shows an example parse tree and the corresponding UD tree:

A quick and handy way to use `gfud` is to pipe its commands to one another:

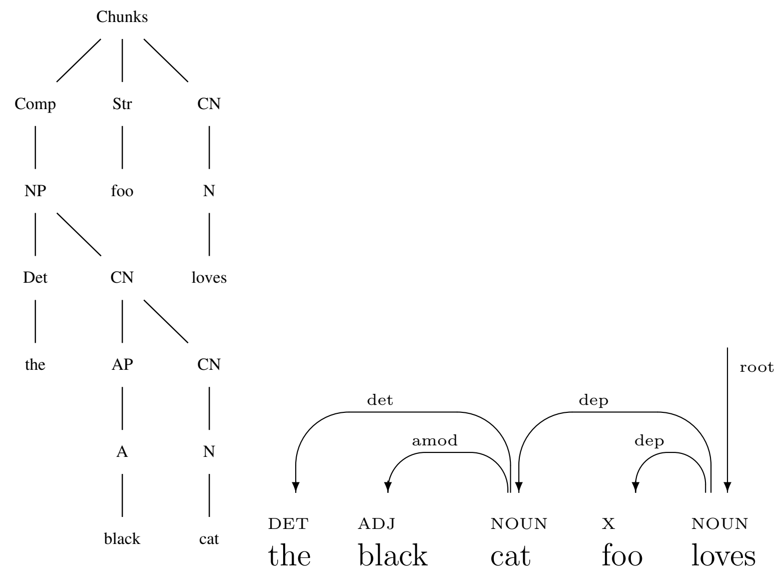


Figure 4.2: An analysis obtained by chunking. The token *foo* is classified as unknown, and no complete tree can be built for the input.

```
$ cat examples.txt | gfud dbnf English.dbnf Utt | gfud conll2pdf
```

This command shows the visualized dependency trees directly.

The `gfud` program has many other functionalities, to which we will return later. Most of them have to do with GF. But there is also a pure dependency parsing related function for evaluation,

```
$ gfud eval (micro|macro) (LAS|UAS) <goldfile> <testablefile>
```

This can be used for evaluating the accuracy of a dependency parser, in a way to be explained in Section 10.9. Here is an example of an experiment:

1. Extract POS-tagged sentences from the English PUD treebank and parse the resulting sentences with English.dbnf, cutting at 100 parses, showing 1 parse per sentence:
   ```
   $ cat eng_pud.conllu | gfud extract-pos-words |
             gfud dbnf English.dbnf -cut=100 -show=1
                                         > my_eng_pud.conllu
   ```
   (this takes 14 seconds on my laptop).

2. Evaluate the result by comparison to the original, using macro-average Labelled Attachment Score

```
$ gfud eval macro LAS eng_pud.conllu my_eng_pud.conllu

UDScore {udScore = 0.5030515854274822, udMatching = 1000,
  udTotalLength = 21183,
  udSamesLength = 10290, udPerfectMatch = 26}
```

Another treebank, the English EWT test treebank, gives 0.61 macro LAS. Interestingly, doing the experiment with a Swedish treebank (the Talbanken test treebank) using the same *English* grammar gives a higher score, 0.54. This gives some hope that the phrase structure analysis given here is interlingually adequate.

These scores are, however, not very high, since the state of the art in UD parsing English and Swedish is over 0.9. The standard method of parsing is based on machine learning from annotated treebanks; the main algorithms will be explained in Chapter 10. Reaching the state of the art uses tens of thousands of trees. To reach LAS over 0.60, a few hundred trees are needed. Writing a DBNF grammar can be a quicker way to reach that level and thereby to bootstrap a parser for a new language. However, going much beyond that level may involve grammar writing work with diminishing returns. A hybrid approach combining machine learning with grammar can be a better way to go.

Chapter 5

The interlingual perspective: words

In this chapter, we will start writing grammars in GF. This will extend the expressivity of grammars in two ways:

- accurate generation of language via precise treatment of morphology and agreement,
- interlingual generalizations via shared syntactic stuctures, **abstract syntax**,
- compact grammar description where a single GF rule can replace hundreds of BNF rules.

GF can be seen as a synthesis of the knowledge we have built up so far: morphology, grammatical functions (as expressed by dependencies), and phrase structure. The goal is to provide tools that enable the reader to approach any language and implement its grammar in GF. As this is a substantial task, we divide the introduction to two chapters. In this chapter, the focus is on words, morphology, and word senses. In the next chapter, we widen it to phrases and syntax.

- Section 5.1 provides the big picture of how GF relates to phrase structure and dependencies.
- Section 5.2 defines the notion of **linearization types**, which is a generalization of words and phrases from strings to inflection tables and other complex objects.
- Section 5.3 discusses the design of linearization types for lexical categories from the point of view of efficiency and effort.
- Section 5.4 introduces the technique of **smart paradigms**, which min-

imizes the effort of lexicon building by the use of linguistic generalizations.

- Section 5.5 goes through the main parts of speech, now from the perspective of accurate generation and interlingual equivalences.
- Section 5.6 explains how an **interlingual lexicon** can be built on the basis of **word senses**.

5.1 From phrase structure to abstract syntax

Abstract syntax is a generalization of phrase structure. It abstracts away from

- the shape of tokens,
- the number of tokens,
- the order of constituents,
- morphological variation,
- the continuity of constituents.

In GF, abstract syntax is expressed in terms of **constructor functions** that operate on **categories**. Categories are a generalization of non-terminals in BNF, and we continue to use the category symbols of the previous chapter. Construction functions, or, briefly, **functions**, can be conceived as names given to BNF rules. Hence the BNF rule

```
  S ::= NP VP
```

is represented as a function named `Pred` (for predication), which takes an NP and a VP as its arguments and returns an S as its value. In GF notation, we write

```
  fun Pred : NP -> VP -> S
```

to introduce this function. The part on the right of the colon (`:`) is the **type** of the function, where NP and VP are **argument types** and S is the **value type**.

The **concrete syntax** gives **linearization functions** for each function in the abstract syntax. For instance,

```
  lin Pred np vp = np ++ vp
```

says that trees formed by `Pred` are linearized so that the NP is **concatenated** (symbol `++`) to VP. The symbols `np` and `vp` are **variables** that stand for the linearizations of the arguments of `Pred`. The choice of their names is just conventional: one could equally well write

```
  lin Pred x y = x ++ y
```

Table 5.1 gives a quick reading guide to GF notation, but it will also be explained in more detail in the text.

To illustrate the power of abstract syntax and linearization, let us consider another rule, where a verb (V) is given an NP complement:

```
  fun Compl : V -> NP -> VP
```

The linearization rule could be

```
  lin Compl v np = v ++ np
```

which together with the above linearization of `Pred` results in the **Subject-Verb-Object** (**SVO**) order of constituents. But changing it to

```
  lin Compl v np = np ++ v
```

results in **Subject-Object-Verb** (SOV), used e.g. in Latin, as well as in German subordinate clauses. A third common order, **Verb-Subject-Object** (VSO) is obtained by the following pair of linearization rules:

```
  lin Pred np vp = vp.verb ++ np ++ vp.obj
  lin Compl v np = {verb = v ; obj = np}
```

This shows VP as a **discontinuous constituent**, modelled as a **record** with two **fields**, recognized by the **labels** `verb` and `obj`. The **projection** operator (`.`) picks a value stored in a record corresponding to a field.

The use of records in addition to strings is one of the two extensions that GF provides over BNF. The other extension is **tables**, which model inflection tables, of the kind we saw in Chapter 2. Tables are used both for words and for complex phrases — words are seen as special case of phrases. For instance, an NP can inflect for case and have inherent **agreement features**. The simplest case is an NP consisting of just one word:

Construct	**Notation**	**Example**
Abstract syntax module	abstract	abstract Lang =...
Concrete syntax module	concrete	concrete LangEng of Lang Lang =...
Resource module	resource	resource ResEng Lang =...
Module extension	**	abstract Lang = Noun,Verb **...
Module opening	open	resource ResEng = open Prelude in...
Abstract syntax category	cat	cat NP
Abstract syntax function	fun	fun Pred : NP -> VP -> S
Linearization type	lincat	lincat N = {s : Number => Str}
Linearization rule	lin	lin Pred np vp = np.s!Nom ++ vp!np.a
Parameter type	param	param Case = Nom | Acc
Auxiliary operation	oper	oper addS : Str -> Str = \x -> x + ‘‘s’’
String concatenation	++	"loves" ++ "Mary"
Token concatenation	+	"Maria" +"m" $\Downarrow$ "Mariam"
Function type	->	NP -> VP -> S
Function application	f a b	Pred np vp
Function abstraction	\ ->	\x,y -> x + y
Table type	=>	Case => Str
Table	table	table {Nom =>"she" ; Acc =>"her"}
Selection from table	!	she_NP.s ! Acc $\Downarrow$ "her"
Table with one branch	\\ =>	\\p,q => np ! q ! p
Record type	{...:...}	{s : Str ; g : Gender}
Record	{...=...}	{s = "doctor" ; g = Fem}
Projection from record	.	{s = "doctor"}.s $\Downarrow$ "doctor"
Record update	**	doctor_N ** {g = Masc}
Case expression	case	case np.a of {Ag n _ => cn.s ! n}
Tuple type	*	Number * Case
Tuple	<,>	<Sg,Dat>
Comment	--	-- comment till the end of line
Comment	{- -}	{- comment of any length -}

Table 5.1: Reading guide for GF notation. The first five rows are about modules, the next six list the different kinds of rules. The rest are expressions for types and objects. The notation $e \Downarrow v$ means that expression e is computed to value v.

```
lin she_NP = {
  s = table {Nom => "she" ; Acc => "her"} ;
  agr = Ag Sg P3
  }
```

A VP can be a table over agreement features, which in predication are passed from the NP to the VP. The subject-verb agreement can then be expressed in a linearization rule as follows:

```
lin Pred np vp = np.s ! Nom ++ vp ! np.agr
```

The **selection** operator (`!`) selects a value from a table, corresponding to a feature. In this example, its first occurrence selects the nominative (`Nom`) of the noun phrase. The second occurrence selects the form of the verb phrase corresponding to the agreement feature stored in the record for the noun phrase. In summary, the rule says that the NP is used in the nominative case and the VP is used in agreement to the NP.

The relation of phrases to dependencies can be maintained in the same way as in DBNF: by assigning a label to each argument of an abstract syntax function:

```
Pred  : NP -> VP -> S  # nsubj head
Compl : V  -> NP -> VP # head obj
```

In contrast to DBNF, these definitions are language-independent: labels are distributed correctly via linarization even when word order varies, so that correct dependency trees can be generated from abstract syntax trees. At the same time, the abstraction creates some complications, to which we will return in Section 6.3.

5.2 Parameters and linearization types

The additional power of GF compared to BNF comes from the use of records, tables, and language-dependent feature types. Feature types are in GF called **parameter types** and defined by rules that enumerate their values, separated by a bar (|):

```
param Case = Nom | Acc
```

Every language can define its own type for case, or also leave it out if it is not needed. Chinese, for instance, does not need case, whereas the Latin case has six values and Finnish 15. Parameter types are used as parts of **linearization types**, which define the inflectional and inherent features of abstract categories (as introduced in Section 2.4). Thus we have in Section 5.1 assumed the following linearization types for English:

```
lincat S  = Str
lincat NP = {s : Case => Str ; agr : Agr}
lincat VP = Agr => Str
```

The keyword **lincat** marks linearization type definitions in GF. The type `Str` means **string** — actually, **token list**, since strings are assumed to be divided into tokens. The double arrow `=>` marks a **table type**, i.e. a the type of tables for a given parameter type. A **record type** looks similar to a record, except that the fields are marked by the colon (`:`) and contain types instead of values. The notation follows the general practice that the colon is used for assigning types, and the equality sign (`=`) to assigning values.

The main principles in GF are the following:

- the **abstract syntax** defines categories (`cat`) and functions (`fun`)
- the **concrete syntax** defines linearization types (`lincat`), linearization functions (`lin`), and parameter types (`param`)
- the type of the linearization function of an abstract function f is formed from the linearization types of the abstract types. More precisely: If

$$\text{fun}\, f \;:\; T_1 \to \ldots \to T_n \to T$$

  then

$$\text{lin}\, f \;:\; \text{lincat}\, T_1 \to \ldots \to \text{lincat}\, T_n \to \text{lincat}\, T$$

- the linearization t^* of an abstract syntax tree t is obtained by computing the linearization function of its outermost function with the linearizations of the subtrees as arguments:

$$(f\, t_1 \ldots t_n)^* \;=\; \text{lin}\, f\, t_1^* \ldots t_n^*$$

The last two principles mean that linearization is mathematically a **homomorphism**, and that it is a **compositional** operator: the linearization of a tree is a function of the linearizations of its immediate subtrees.

The mathematical properties of GF are important for its efficient implementation, but they also impose a restriction on how much exactly can be done in linearization. A tree can be linearized in many different ways in different languages, but not in all conceivable ways in which trees can be converted to strings. This will sometimes impose a restriction on what the grammar programmer can do. But it can also be interesting for the linguist to get a formal definition of what interlingual structures are compositional, as well as what translation equivalences are *not* compositional.

Another practical consequence of the discipline imposed by linearization types in GF is that grammars are **statically checked**. The GF compiler guarantees that all features are consistently used and that linearization never fails at runtime. Static type checking is a general feature of programming languages such as Java and Haskell, in contrast to Python or LISP. Static checking is not common in grammar formalisms, let alone informal annotation schemes such as UD. For the working grammarian, static checking is helpful, because the compiler finds many bugs before the grammar is put to use. But some programmers — for instance those used to Python — may find it annoying, as it delays the point at which one can start testing the grammar.

5.3 Linearization types for lexical categories

With the ultimate goal of building an interlingual grammar, let us start bottom-up, from words, as we did earlier before proceeding to syntactic dependencies and phrase structure. A good place to start is nouns, for which we will use the category name `N`. We will include it as the first item in an **abstract syntax module** called `Example` and shown in Figure 5.1. Beside it, we show a **concrete syntax module** called `ExampleEng`, which defines how trees built in `Example` are linearized in English.

Below the abstract and concrete syntax we have a **resource module** called `ResEng`, which defines parameter types and **auxiliary operations** (`oper`) that are usable in concrete syntax modules via a directive to `open` it.

It is good practice to include all `param` and `oper` definitions in `resource` modules, so that they can be reused in different concrete syntaxes. Thus we will start our discussion of morphology by working inside a resource module.

Another good practice is to introduce **type synonyms**, such as `Noun` in `ResEng`, and use them instead of explicit tables and records. This is less

```
abstract Example = {

cat
  N ;
fun
  doctor_N : N ;
}
```

```
concrete ExampleEng of Example =
  open ResEng, Prelude in {
lincat
  N = Noun ;
lin
  doctor_N = regNoun "doctor" ;
}
```

```
resource ResEng = {
param Number = Sg | Pl ;
oper
  Noun : Type = {s : Number => Str} ;
  mkNoun : (sg,pl : Str) -> Noun
    = \man,men -> {s = table {Sg => man ; Pl => men}} ;
  regNoun : Str -> Noun
    = \dog -> mkNoun dog (dog + "s") ;
}
```

The GF compiler assumes that each module resides in a file called the module name plus the suffix `.gf`. After starting the GF shell with the command `gf`, the following commands are useful for testing the grammars:

```
> i ExampleEng.gf      -- to import a concrete syntax
> l doctor_N           -- to linearize a tree
> p -cat=N "doctor"    -- to parse a string

> i -retain ResEng.gf  -- to import a resource
> cc regNoun "baby"    -- to compute an expression
```

Figure 5.1: Abstract and concrete syntax modules, as well as a resource module that is used in the concrete syntax module. The module `Prelude` belongs to the standard library of GF and is often useful to open.

verbose, as seen in the types of `mkNoun` and `regNoun`, where it also guarantees that the same type is targeted in both of them. But it also makes it possible to change the type of nouns without changing much of the code. For instance, if a later version of the grammar needs to add gender information to English nouns — for instance, to decide whether the reflexive pronoun is *himself*, *herself*, or *itself* — the only change required is in the definitions of `Noun` and `mkNoun`:

```
  Noun : Type = {s : Number => Str ; g : Gender}
  mkNoun : (sg,pl : Str) -> Noun
    = \man,men -> {s = table {Sg => man ; Pl => men} ; g = Neutr} ;
```

Since these are the only places where the actual record-table structure of nouns is shown, the rest of the code can remain intact. Of course, the gender field will have to be changed for those nouns where the default `Neutr` is not the correct choice. This can be done either by an operation that sets the gender of a noun,

```
  oper setGender : Gender -> Noun -> Noun
    = \gen,noun -> noun ** {g = gen} ;

  lin doctor_N = setGender Fem (regNoun "doctor") ;
```

or by explicitly overriding the default in the lexicon,

```
  lin doctor_N = regNoun "doctor" ** {g = Fem} ;
```

In both cases, the **record update** operator `**` is used to overwrite an earlier value in a record. If the field is missing, this operator adds it to the record.

The `mkNoun` operation is the **worst-case paradigm** for nouns. It lists all those forms that can be different for an object of type `Noun`. Worst-case paradigms are natural companions of type synonym, making it possible to change the underlying table and record types without affecting most of the code. In programming terms, this means that we treat the linearization types as **abstract data types**.

As we saw in Chapter 2, a good practice in a morphological lexicon is to keep the size of the inflection table minimal. In GF, this means keeping the linearization type size equal to the number of arguments of the worst-case paradigm.

One thing that sometimes obscures the view of the number of inflection forms required is their use in syntactic combinations. For example, even though common nouns in English do not have accusative cases, noun phrases (NP) in general do, because noun phrases also include personal pronouns (Pron). The linearization type a category must always have room for the "worst case", and therefore noun phrases must inflect for the two cases:

```
lincat NP = {s : Case => Str ; a : Agr}
```

We will discuss the formation of complex phrases in next chapter, so let us just look at a special case: a function that forms the singular definite noun phrase from a noun, e.g. *the doctor* from *doctor*:

```
fun TheSg : N -> NP
lin TheSg n = {s = \\c => "the" ++ n.s ! Sg ; a = Ag Sg P3}
```

The accusative form of these noun phrases is hence identical to the nominative, and there is no need for the noun itself to make case distinctions.

The example with English noun case may sound trivial, but actually a similar thing has been debated in the grammar of Finnish for a long time. In Finnish, nouns have 14 inflectional cases, whereas pronouns have 15: the same as nouns plus an accusative case. When a noun phrase is used in a context where the accusative is needed (as a direct object of a verb), a pronoun NP is rendered in the accusative, and a noun NP either as nominative or genitive, depending on the syntactic context. Some morphological analysers then give accusative as an alternative analysis every time they encounter a nominative or genitive noun, but this is of course redundants. The standard UD treebanks use a POS tagger that use the `Case=Acc` feature only for pronouns.

However, for English verbs, the practice in UD treebanks is different. Thus the infinitive form of a verb can receive one of the features

```
Mood=Ind|Tense=Pres|VerbForm=Fin
VerbForm=Inf
```

depending on the syntactic context. The distinction probably comes from the POS tagger, which takes some syntactic aspects into account; for the morphological analyser as such, there is no reason to make the distinction. Notice that infinitive verb forms could also have the tag

```
Mood=Ind|Number=Sing|Person=1|Tense=Pres|VerbForm=Fin
```

but this is in UD used only for the verb form *am*.

In syntax, both agreement and tense have to be taken into account when verb phrases (VP) are formed. Then the relevant parameter types are

```
Aspect = Simple | Perf
Tense  = Pres | Past | Fut
Number = Sg | Pl
Person = P1 | P2 | P3
```

The product of these types leads to 36 "VP forms" ("$2 \times 3 \times 2 \times 3$), and then we have not yet included infinitive, imperative conditional, and progressive forms. Verbs themselves, however, need only five morphological forms:

```
VForm = Inf | PresSgP3 | PastInd | PastPart | PresPart
```

When a verb prase is formed, these can be captured by a compact pattern matching:

```
oper verbForm :
  Verb -> Aspect -> Tense -> Number -> Person -> Str
    = \a,t,n,p -> case <a,t,n,p> of {
    <Simple, Pres, Sg, P3> => v.s ! PresSgP3 ;
    <Simple, Pres, _,  _ > => v.s ! Inf ;
    <Simple, Past, _,  _ > => v.s ! PastInd ;
    <Simple, Fut,  _,  _ > => "will" ++ v.s ! Inf ;
    <Perf,   Pres, Sg, P3> => "has"  ++ v.s ! PastPart ;
    <Perf,   Pres, _,  _ > => "have" ++ v.s ! PastPart ;
    <Perf,   Past, _,  _ > => "had"  ++ v.s ! PastPart ;
    <Perf,   Fut,  _,  _ > => "will have" ++ v.s ! PastPart
    }
```

Notice the use of pattern matching over a tuple, with disjunctive (|) and wildcard (_) patterns (see Figure 5.2 for more examples of pattern matching).

Another example we considered in Chapter 2 were Latin nouns. We noticed in that at most 10 forms are different for any given noun, even though the product of two numbers and six cases is 12. If we really want to optimize the table in this way, we have two alternatives: hierarchic parameter types and records.

Hierarchic parameter types work technically like **algebraic datatypes** in programming, where the parameter constructors can take arguments from other types. If we start with the baseline of two numbers and six cases for Latin,

```
  Number = Sg | Pl
  Case   = Nom | Acc | Gen | Dat | Abl | Voc
```

we end up with tables of the type

```
  Number => Case => Str
```

for nouns, which contain 12 forms, where the plural dative and ablative are always the same, and so are the plural vocative and nominative. To eliminate this redundancy, we can use the following system of types:

```
  Number   = Sg | Pl
  Case     = Nom | Acc | Dat | Gen
  NounForm = NF Number Case | NSgAbl | NSgVoc
```

The type `NounForm` now has exactly $10 = 2 \times 4 + 1 + 1$ values. This may be a bit of overkill, since it complicates the formulation of agreement rules later in syntax. But it makes more sense in cases where more forms can be saved.

An instructive example is German adjectives, which inflect for number, gender, and case. In addition, they inflect for degrees (three values) and the "strong", "weak", and "mixed" declensions. The cross-product of these features has 216 forms:

$$3\text{degrees} \times 3\text{strenghts} \times 2\text{numbers} \times 3\text{genders} \times 4\text{cases}$$

All these forms, plus some more for predicative uses, can be found e.g. in the tables in the Wiktionary

`https://de.wiktionary.org/wiki/Flexion:schlimm`

Just like English verb phrases, these distinctions are relevant when forming adjectival phrases (AP) in German. But in morphology, we can minimize the number of forms by the following observations:

- the forms in weak and mixed declensions can be found from the strong forms
- gender matters only in the singular number (this is so in the Wiktionary, too)
- the accusative is the same as the nominative except for masculine singular
- the masculine and neuter dative are always the same
- and some more observarions, ending up with the conclusion that there are only 6 different forms in each degree, exemplified by *schlimm, schlimme, schlimm**em**, schlimm**en**, schlimm**er**, schlimm**es***

An algebraic datatype could be used to capture the German adjective inflection precisely. But it would be very complex and uniniituitive, in comparison to a record, where the forms are named after the most typical carrier of a certain ending:

```
AdjForms : Type = {
  pred, mnom, macc, mdat, fnom, nnom : Str
  }
```

This reduces the number of adjective forms that need to be given by the grammarian and stored in the run-time system to 18. The "complete" set of forms needed by the AP can be easily computed by a pattern matching similar to the one showed for English VP above.

The most extreme saving in the size of inflection tables is achieved by storing only a set of different **stems**, from which the actual inflection forms are obtained by **agglutination** — pure concatenation of endings. Since German adjectives are always have the same set of 6 endings, once the degree form is given, they could be defined by the following minimal type, which contains just four strings:

```
MinimalAdjForms : Type = {
  pospred, posstem,
  compstem, supstem : Str
  }
```

(In the positive degree, the predicative form may end with an *e*, in which case the stem is withour *e*.) To construct all the 18 inflection forms, all that is needed is to glue proper endings to proper stems. If done in the syntax part of the grammar (when AP is formed), the function `Predef.BIND` must be used instead of the `+` operator:

```
  mnom = a.posstem ++ Predef.BIND ++ "er"
```

an so on. The reason for this is technical: the GF compiler must know all tokens at compile time, and the + operator would build new tokens at runtime. This makes the use of this approach a bit cumbersome, and it is in the RGL only done in Finnish, Maltese, and Turkish, where the inflection tables would otherwise grow to the size of hundreds or thousands of forms.

5.4 Smart paradigms

The previous section was about designing the linearization types of words, to optimize the storage needed for them in the compiled grammar. In this section, we will look at how to minimize the grammarian's effort when creating them, or, if the lexicon is extracted automatically, the amount of data that is needed to derive the full inflection.

We have already shown a very simple case of this in Figure 5.1: the operation `regNoun`, which builds the plural form from the singular by adding an "s". For those nouns that do not follow this pattern, two forms must be given by using `mkNoun`. However, `regNoun` can be made smarter by **pattern matching** on the singular form, as shown in Figure 5.2.

Similar paradigms as in Figure 5.4 could be easily defined for English verbs and adjectives. The leading principle is that inflection is inferred from as few characteristic forms as possible. An evaluation of smart paradigms in the GF RGL (Détrez and Ranta 2012) showed that less than two forms on the average were typically enough, even for Finnish, which has a rich morphology.

For a language that has an established standard paradigm system, like Latin, it makes sense to implement each paradigm as a separate function and let the smart paradigms decide which standard paradigm to use. In addition to the characteristic forms, the gender is typically a useful feature to pattern-match on. The following paradigm using the singular nominative and genitive, together with the gender, gives a good coverage of Latin nouns:

```
  smartNoun : (nom,gen : Str) -> Gender -> Noun
    = \nom,gen -> case <nom,gen> of {
      <_ + "a",              _ + "ae"> => decl_I_Noun nom g ;
      <_ + ("us"|"er"|"um"), _ + "i" > => decl_II_Noun nom g ;
      <_ + ("us"|"u"),       _ + "us"> => decl_IV_Noun nom g ;
  smartNoun : Str -> Noun
    = \s -> case s of {
        _ + ("a"|"e"|"o"|"u") + "y" => regNoun s ;
        x + "y"                     => mkNoun s (x + "ies") ;
        _ + ("s"|"z"|"x"|"sh"|"ch") => mkNoun s (s + "es") ;
        _ => regNoun s
      } ;

  mkN = overload {
    mkN : Str -> Noun
      = \s -> smartNoun s ;
    mkN : (sg,pl : Str) -> Noun
      = \sg,pl -> mkNoun sg pl ;
    } ;
```

Key GF concepts used:

- a **case expression** for pattern matching over strings
- **regular patterns** in case expressions, where
  - a double-quoted string matches itself
  - _ matches anything
  - a variable (e.g. `x`) matches anything and can be used on the right hand side of `=>`
  - a disjunctive pattern (`|`) matches alternatives
  - a sequence pattern (`+`) matches sequences
- an **overloaded operation**, where
  - different operations can be given the same name
  - these operations just have to have different types
  - the GF grammar compiler **resolves** the overloading based on the type

Figure 5.2: A smart paradigm and an overloaded paradigm group.

```
    <_ + "es",              _ + "ei"> => decl_V_Noun nom g ;
    <_,                     _ + "is"> => decl_III_Noun nom gen g ;
    _ => Predef.error ("smartNoun doesn't match" ++ nom ++ gen)
    }
```

Some observations from this paradigm:

- `decl_II_Noun` and `decl_IV_Noun` make themselves pattern matching over the ending
- `decl_III_Noun` has no particular nominative ending or gender, but is recognized from the genitive ending
- the last case captures nouns that do not fit into any of the declensions, either due to a typo when using the paradigm or because of irregularity

Another type of standard linguistic knowledge that helps in paradigm definitions is **phonological and orthographical variation**. An example is **consonant duplication** in English. It means that certain final consonants are duplicated in certain verb and adjective forms, when appearing after a single vowel. Figure 5.3 shows the definition of consonant duplication as a string operation and its use in verb and adjective paradigms. The other special cases of these paradigms (such as basic forms ending in *e* or *y*) are omitted.

The above examples taken from English, German, and Latin cover many of the variations that can be found in different languages and will hopefully help in implementing their morphologies. One important type of morphology that we have not discussed here is the **non-concatenative morphology** of Semitic languages such as Arabic and Maltese. The key idea for dealing with it is to use records of three strings, instead of simple strings, to represent **roots** of words. Each field in the string contains a **radical**, usually one consonant. Inflectional forms are built by combining the root record with a **FaCaL pattern**, which is a record of four strings:

```
Root    : Type = {F,C,L : Str}
Pattern : Type = {F,FC,CL,L : Str}
fill : Pattern -> Root -> Str
  = \p,r -> p.F + r.F + p.FC + r.C + p.CL + r.L + p.L
```

Full details can be found in the GF book (Section 4.5) and in the corresponding section in the on-line GF tutorial, and of course the RGL code for Arabic and Maltese.

```
  vowel : pattern Str
    = #("a" | "e" | "i" | "o" | "u") ;

  consonant : pattern Str
    = #("b" |"d" |"g" |"l" |"m" |"n" |"p" |"r" |"t" |"z") ;

  duplFinalCons : Str -> Str = \s -> case s of {
    _ + #vowel + #vowel + ?     => s ;
    _ + #vowel + c@(#consonant) => s + c ;
    _ + #vowel + "c"            => s + "k" ;
    _                           => s
    } ;

  regVerb : Str -> Verb = \s ->
    let
      ss = duplFinalCons s
    in
    mkVerb s (s + "s") (ss + "ed") (ss + "ed") (ss + "ing") ;

  regAdj : Str -> Adj = \s ->
    let
      ss = duplFinalCons s
    in
    mkAdj s (ss + "er") (ss + "est") ;
```

Key GF concepts used:

- **pattern macros**, `vowel` and `consonant`, formed by prefixing `#` to a pattern
- use of pattern macro in pattern matching, `#vowel`
- the **single character pattern** `?` matching a single-character string
- the **alias pattern** `c@#consonant`, which binds the variable `c` to the value matched by the pattern `#consonant`
- the **local definition** (`let`... `in`) defining local constants, typically in the scope of variables

Figure 5.3: English consonant duplication, as an example of string operations and their use in paradigms. Other special cases of these paradigms have been omitted.

5.5 Parts of speech revisited

In Table 4.1, we listed the lexical categories used in the phrase structure rules of Chapter 4. The same categories are used in the GF Resource Grammar Librarary, and their full list is given later, in Table **??**. In this section, we will give some guidelines about their linearization types in different languages, thereby formalizing the discussion in Chapter 2.

Nouns, as we have seen, typically inflect for number and case (if the language has these features) and have an inherent gender (if the language has genders). The can also inflect for **definiteness**, as in Scandinavian languages. In Chinese and Thai, the **classifier** can be included as an extra string:

```
lincat N =
  {s : Number => Str}                               -- Eng
  {s : Number => Str ; g : Gender}                  -- Fre, Ita
  {s : Number => Case => Str ; g : Gender}          -- Lat, Ger
  {s : Number => Def => Case => Str ; g : Gender} -- Swe, Ice
  {s : Str ; c : Str}                               -- Chi, Tha
```

Proper names are like numbers but without number variation. They typically behave like singular noun phrases in agreement, but may need an inherent number to account for group names (*The Beatles*) or company names (*Google increase their revenues*).

```
lincat PN =
  {s : Case => Str ; g : Gender, n : Number}
```

Adjectives can be "3 times 3 times nouns" as in Latin, where each of the three degrees produces noun-like inflection tables for each of the three genders. In German, we can used the optimized `AdjForms` record type defined in Section 5.3. In English, only the degree matters. But an inherent feature `isComp` can be added to determine if the adjective is complex, i.e. if its comparison is formed syntactically by the adverb *more* and *most*; an alternative is to include the adverb in the table itself, but this is not elegent from the morphological point of view. In Chinese, a similar feature can be used to say whether the adjective needs the particle *de* when used as an attribute. In many languages, an `adv` field might be added to tell how adverbs are formed from the adjective.

```
  lincat A =
    {s : Degree => Str ; isComp : Bool}            -- Eng
    {s : Degree => Gender => Number => Case => Str} -- Lat
    {pos,comp,sup : AdjForms}                      -- Ger
    {s : Str ; isComp : Bool}                      -- Chi

    ** {adv : Degree => Str}            -- many languages
```

Adverbs in general can be formed from adjectives (*warmly*) but also independently (*now*). When formed from adjectives, adverbs typically have degrees parallel to the adjective degrees. But in general, adverbs do not have degrees, and their linearization type is hence just a string. In addition, there can be inherent features, such as in Chinese a feature saying whether the adverb expresses time, place, or manner; this feature is needed to decide the proper place of the adverb in a sentence, e.g. whether it is before or after the verb:

```
  lincat Adv =
    {s : Str}                -- Eng, Fre, Ger,...
    {s : Str ; t : AdvType}  -- Chi
```

The adverb type feature belongs to the concrete syntax and should not be confused with another distinction, which is needed already in the abstract syntax: separate categories for

- adverbs modifying adjectives, `AdA` e.g. *very*
- adverbs modifying numerals, `AdN` e.g. *almost*
- sentential adverbs, `AdV` e.g. *never*
- interrogative adverbs, `IAdv` e.g. *why*

The difference is that adverbs of different abstract categories have different possible combinations, not just different positions in linearization. This for instance an Adv can be used in the predicative position, but an AdA cannot:

the meeting is now

**the meeting is very*

However, the RGL classification of adverbs is one of the least exact regions of the library. Some adverbial words can be used in many of these ways, which means that they need to have separate abstract syntax functions in the categories involved. At the same time, the main category Adv allows dubious combinations, such as *the meeting is warmly*. This would motivate finer distinctions, which are of course possible to do in semantic grammars, to be discussed in Chapter 8.

Verbs are in many languages the most complex category. They have a rich morphology, but also a wide variety of combination possibilities due to verb valencies. As for morphology, it is important to keep the number of forms down to the absolutely necessary, i.e. only to include inflectional features that can produce different forms. Thus for instance compound tenses (*have walked*, *will walk*) should not be included, since they can be formed from participles and infinitives. We concluded in Chapter 2 that English verbs need five forms, which are conveniently listed in a special parameter type `VForm`. In Romance languages and Latin, a hierarchic system of algebraic datatypes is natural. Thus in Latin, the following system is an accurate formalization of how verb inflection is structured in traditional grammars, and also the most economic way to store all and only the different verb forms:

```
VForm = VFAct VActForm | VFPass VPassForm
      | VFInf VInfForm | VFImp VImpForm
      | VFGer VGerund  | VFSup VSupine
      | VFPart VPartForm ;
VActForm  = VAct VAnter VTense Number Person ;
VPassForm = VPass VTense Number Person ;
VInfForm  = VInfActPres | VInfActPerf Gender | VInfActFut Gender
          | VInfPassPres | VInfPassPerf Gender | VinfPassFut ;
VImpForm  = VImp1 Number | VImp2 Number Person ;
VGerund   = VGenAcc | VGenGen |VGenDat | VGenAbl ;
VSupine   = VSupAcc | VSupAbl ;
VPartForm = VActPres | VActFut | VPassPerf ;
VAnter    = VAnt | VSim ;
VTense    = VPres VMood | VImpf VMood | VFut ;
VMood     = VInd | VConj ;
```

We leave it as an exercise to calculate the number of values of tyoe `VForm` :-) For Latin verbs, unlike German adjectives, all of these forms, which are traditionally included in inflection tables, can actually be different and are therefore relevant.

Verbs may also have inherent features, for instance,

- whether the verb is reflexive or deponent (used in passive only)
- what auxiliary is used for compound tenses (in e.g. French, Italian, and German)

Let us collect these features in the parameter `VType`. One more thing we need is **particles** used in compound verbs, e.g. *look up* in English, where *up* is a separate word, or *auf+passen* in German, where *auf* is sometimes separate, sometimes a prefix glued to the verb. From these considerations, we end up with a general form of a type synonym,

```
Verb = {s : VForm => Str ; p : Particle ; t : VType}
```

On top of this, we can build linearization types for different **subcategories** of verbs, encoding their **valencies**:

- `V`, one-place verbs, such as *sleep*:
  ```
  lincat V = Verb
  ```
- `V2`, two-place verbs, such as *love*:
  ```
  lincat V2 = Verb ** {c : ComplCase}
  ```
  where `ComplCase` can include a preposition in addition to case.
- `V3`, three-place verbs, such as *give*:
  ```
  lincat V3 = Verb ** {c1,c2 : ComplCase}
  ```
- `VS`, sentence-complement verbs, such as *believe*:
  ```
  lincat VS = Verb ** {m : Mood}
  ```
  The mood parameter is needed in languages such as French, where the mood of the subordinate clause (indicative or subjunctive) depends on the verb.
- `VV`, verb-phrase-complement verbs, such as *want*:
  ```
  lincat VV = Verb ** {i : InfType}
  ```
  where `InfType` records the form expected from the VP complement (e.g. plain infinitive in *can sing*, infinitive with *to* in *want to sing*, and gerund in *start singing.*
- `V2V`, NP+VP-complement verbs, such as *order*:
  ```
  lincat V2V = Verb **
    x{c : ComplCase ; i : InfType ; oc : Bool}
  ```
  where `oc` records whether the VP complements agrees to the object (e.g. *I order you to wash yourself* or the subject *I promise you to wash myself*).

Verb valencies in the RGL thus have two aspects:

- abstract **logical type**: which categories serve as complements

- concrete **rection**: what forms (e.g. case, mood, infinitive form) the complements take

Distinguishing between these aspects of valency is a crucial idea in an interlingual lexicon: the abstract syntax encodes the logical types of verbs, whereas rection is language-dependent. For example, whether a two-place verb is transitive (takes an direct object, `obj` in UD) is not specified in the abstract syntax, because an equivalent verb in another language may take a prepositional object (`obl`) in UD. In this way *regarder* in French can have the same abstract syntax entry as English *look at*.

More subcategories of verbs will be introduced in Chapter 8. Let us conclude this section with some categories of **function words** and their linearization types. Function words, also known as **structural words**, are typically very few in each category. Nevertheless, they require a lot of attention from the grammarian, since they often have complex and irregular morphology. They may also have more inflection forms than content words: for instance, pronouns in English have the accusative case, which nouns and proper names do not have.

Personal pronouns (`Pron`) are typically inflected for case and have inherent agreement features (gender, number, person). In the RGL, each pronoun record includes the corresponding **possessive pronoun**, with alternative forms much like a determiner (see below), since the use of possessive pronouns is syntactically like determiners.

```
  Pron = {s : Case => Str ; poss : Determiner ; a : Agr}
```

In Enlish, pronouns need two case forms (*I, me*) and two determiner forms (*my, mine*). In German, where determiners inflect similarly to adjectives (but without degrees), 16 forms are needed for the possessive and four case forms for the pronoun itself. Some languages, e.g. Romance and Slavic, make a distinction between **stressed and unstressed pronouns**, where the latter may appear as **clitics** or even be realized as empty strings because of **pro-drop**. An extensive example with Italian can be found in the GF book, Chapter 9, and the on-line GF tutorial.

Conjunctions (`Conj`) may need two strings (like *both* - *and*):

```
  Conj = {s1, s2 : Str}
```

Prepositions typically need a string and an inherent case. In the RGL, they actually correspond to **adpositions**, and may need an inherent feature indicating if the adposition is used as s preposition or a postposition. Alternatively, they can have two strings, one before and one after the phrase they are attached to, and one of these may be empty. If both fields are empty, the abstact adposition is linearized to just a case passed to the noun phrase:

```
Prep = {s1, s2 : Str ; c : Case}
```

This type is typically the same as the complement case specified for verbs.

Determiners (`Det`) inflect much like adjectives, except that they have no degrees and their number is inherent:

```
Det = {s : Gender => Case => : Str ; n : Number}
```

A special category in the RGL is called **quantifiers** (`Quant`), which have both singular and plural forms. An example is English *this-these*. But also articles (*the*, *a*) and possessive pronouns are from an interlingual perspective in this category, since they can be combined with both singular and plural nouns. In many languages, they consistently have different forms for different numbers.

```
Quant = {s : Number => Gender => Case => : Str}
```

We will introduce other function word categories later when discussing syntax. A typical feature for many of them is that they have no inflection. Therefore the syntactic part of speech criterion is the only way to distinguish them from each other.

5.6 Interlingual lexicon and word senses

The abstract syntax of an interlingual lexicon is a collection of **word senses**. Word senses are defined in terms of **translation equivalence**. Their purpose is to support **compositional translation**, which in GF means parsing the source language with its concrete syntax and generating the resulting abstract tree with the concrete syntax of the target language. The abstract tree remains the same, which means that the translation is **structure by structure** and, ultimately, when the leaves of the tree are reached, **word by word**.

Word senses are needed in the abstract syntax to encode **ambiguities** that may arise when a word can have several translations with different meanings. The RGL has some sense distinctions built in for structural words:

`youSg_Pron`, singular *you*, *du* in German

`youPl_Pron`, plural *you*, *ihr* in German

`youPol_Pron`, polite *you*, *Sie* in German

But these distinctions need to be extended to gender-specific forms, such as *anta* (masculine singular) vs. *anti* (feminini singular) in Arabic, with similar distinctions in the plural and moreover in the dual number. So a rough estimate of the number of senses of *you* is 12 (3 numbers, 2 genders, 2 politeness levels). Few languages have the full set as separate words, but the gender distinction may still be needed in the inherent gender used in agreement: French *tu es heureux* ("you are happy") uses the masculine singular *you*, whereas the feminine singular produces *tu es heureuse*.

A large-scale collection of interlingual senses is taking place in the GF-WordNet project,

`https://cloud.grammaticalframework.org/wordnet/`

to which we will return later in Chapter 8. It is based on the Princeton WordNet, which is a database of English word senses. The Princeton WordNet has been adapted to several languages, sometimes directly translated. Analysing the translations has revealed that the original WordNet senses are not directly usable as a translation interlingua; this was not their original goal either. Some sense distinctions in the WordNet are too fine-grained and irrelevant for translation in any language, whereas some senses are not fine-grained enough. For example, the word *drug* has as one of its senses "medical or narcotic substance", but many languages have distinct words for medical and narcotic drugs.

Another challenge for word-to-word translation is that one often needs **multiword constructions** to translate single words. Luckily, these can in GF be encoded as abstract syntax functions and given precise linearization rules to support compositional translation in a more abstract sense. We will return to this question in Chapter 8.

Chapter 6

The interlingual perspective: syntax (IN PROGRESS)

In this chapter, we will develop an abstract and concrete syntax for phrases. We will follow roughly the same order as in Chapter 4 and see how an abstract syntax is extracted from phrase structure. But we will be more careful now: we cannot just keep adding rules to cover more ground, as we must make sure to obey the restrictions imposed by linearization types. This discipline has advantages as well:

- once we know the linearization types, the linearization rules almost write themselves automatically,
- we can build a large grammar with much fewer rules than when using BNF.

The main reason of the discipline is, of course, that we will now be able to build a grammar that can accurately generate language and use a shared interlingual structure. Parsing is also possible with the grammar, but it is less robust than with the overgenerating BNF grammar.

The description below aims at a good coverage of syntactic structures and is therefore quite advanced. A more accessible introduction can be found in the "mini resource grammar" description in the GF book, Chapter 9, and the corresponding part of the book slides

```
http://www.grammaticalframework.org/gf-book/gf-book-slides.
pdf
```

starting from p. 366. To minimize overlap with the GF book, we are not repeating the mini resource here but go directly to the "real thing". Moreover, we will make it on a generic level ready for typical variations in many languages, which means that even the concrete syntax is somewhat abstract. This is done in the same way as in the previous chapter: we will for instance say that noun prases inflect for case without specifying what the cases are. More details of individual languages can be found in chapter 7.

6.1 General principles

We introduced in Section 5.1 the concepts of functions and their linearizations, which were later in Chapter 5 applied to lexical categories, that is, to abstract syntax functions that take no arguments. The general idea was that each word is linearized into a record that consists of an inflection table and inherent features, as well as possibly discontinuous parts, such as with particle verbs (*look* - *up*) and conjunctions (*both* - *and*).

Exactly the same idea applies to complex phrases: they are linearized to records of similar types as lexical items. Thus a noun modified by an adjective linearizes to a table of the same shape as a single noun. For instance, Latin *rosa delectabilis* ("delightful rose") is inflected like the noun *rosa* followed by the adjective *delectabilis* in the feminine gender and in the same number and case as the noun:

	singular	plural
nominative	*ros**a** delectabil**is***	*ros**ae** delectabil**es***
accusative	*ros**am** delectabil**em***	*ros**as** delectabil**es***
genitive	*ros**ae** delectabil**is***	*ros**arum** delectabil**ium***
dative	*ros**ae** delectabil**i***	*ros**is** delectabil**ibus***
ablative	*ros**a** delectabil**i***	*ros**is** delectabil**ibus***

This reflects the idea of the syntactic substitution test: a complex CN can fit in the same places as a simple N.

The rule that produces the inflection table of *rosa delectabilis* is the adjectival modification function and its linearization:

```
fun AdjCN : AP -> CN -> CN
lin AdjCN = {
  s = \\n,c => cn.s ! n ! c ++ ap.s ! cn.g ! n ! c ;
  g = cn.g ;
  }
```

The linearization type of CN is the same as the type of N, whereas the AP is different from A, as it has a fixed degree:

```
lincat N, CN = {s : Number => Case => Str ; g : Gender}
lincat A  = {s : Degree => Gender => Number => Case => Str}
lincat AP = {s :           Gender => Number => Case => Str}
```

Trees of phrasal categories are ultimately formed from their **head lexical items** by **lexical insertion functions**:

```
fun UseN : N -> CN
lin UseN n = cn

fun PositA : A -> AP
lin PositA a = {s = a.s ! Posit}
```

The linearization of lexical insertion can be simply an identity function, like in `UseN` here. But it can also be a bit more more complex, as it for instance fixes some features, like the degree in `PositA`. In such cases, there are typically many lexical insertion functions: for A to AP, we also have the choices of comparative and superlative forms, to be discussed in Section 6.1.6 below. But even then, the linearization rules will be relatively simple.

In BNF, the adjectival modification rule for Latin would be simply

```
CN ::= CN AP
```

assuming the post-modifier order; there can be another rule where the AP precedes the CN. This rule, however, is vastly overgenerating: it permits the combination of a CN of any gender and in any number and case with an AP in any gender, number, and case. This may be fine when the grammar is used for analysing language. But if we want to use the grammar for generation, we must encode the agreement rules.

In BNF, agreement can be expressed by duplicating the categories for every value of inflectional and inherent features. This would result in 30 $(= 3 \times 2 \times 5)$ rules of the shape

```
CN_fem_sg_nom ::= CN_fem_sg_nom AP_fem_sg_nom
```

Such rules would of course be tedious to write. What is more, they would be extremely language-specific and obscure the interlingual perspective.

Another feature of BNF rules that obscures the interlingual perspective is the explicit use of **syncategorematic words**, that is, words that appear "between categories" and make no semantic contribution of their own. We have encoded such words in BNF by non-terminals starting with small letters; the copula is a typical example:

```
  VP ::= cop AP
```

In this chapter, we will get rid of all syncategorematic words in the abstract syntax, and introduce them — in language-specific ways — in concrete syntax. The most extreme example is perhaps the cluster of auxiliary verbs and negation in English sentences:

```
  S ::= NP aux? have? do? neg? VP
```

Depending on the usage of the sentence, the syncategorematic elements may also appear before the subject NP, and their very appearance depends on whether the VP itself has an auxiliary. In GF, this rule is split into two abstract syntax functions: one forming a **clause** (Cl) and another using the clause as a sentence with different temporal features (Temp) and polarities (Pol).

```
  fun PredVP : NP -> VP -> Cl
  fun UseCl  : Temp -> Pol -> Cl -> S
```

The linearization of these functions is the most complex part of the concrete syntax of most languages (see Section 6.2.2). But keeping the abstract syntax as simple as this helps us maintain the interlingual perspective that will be essential in semantics (Chapter 8) and translation (Chapter 9).

6.2 Phrasal categories and their construction functions

We will follow the same order of presentation as in Section 4.3. We will start each subsection with the relevant categories and their linearization types, which are given in a schematic way, using typical inflectional and inherent features but not going to details of what values those features take in different languages. Chapter 7 will go deeper with the details of some languages, as well as exceptions from the schematic patterns.

For the functions, we will show function declarations and their schematic linearization rules following the schematic linearization types. We will also show dependency labels using the same notation as in DBNF: a hash mark followed by the sequence of labels — for example,

```
fun PredVP : NP -> VP -> Cl # nsubj head
```

The number of labels must match the number of argument types, and exactly one of them must be `head`. This will take care of most of the desired labels in a language-independent way. However, the labels corresponding to syncategorematics words (such as `cop`) will not appear among these labels, and have to be added by language-specific annotations, to be discussed in Section 6.3.

6.2.1 Utterances and texts

The RGL distinguishes between utterances and **texts**, where punctuation is introduced only when an utterance is converted to a text:

```
cat Text ; Utt ; Punct
lincat Text, Utt, Punct = {s : Str}

fun mkText : Utt -> Punct -> Text # head punct
lin mkText utt punct = {s = utt.s ++ punct.s}
```

Even these rules have variations: for instance, in Spanish, the question mark is discontinuous, with parts concatenated to both sides of the utterance:

```
lin questionMarkPunct = {s1 = "¿" ; s2 = "?"}
```

The function `mkText`, like all functions with names starting with a small letter, are not ultimate constructors in the RGL, but shortcuts to more detailed rules available in the RGL API. In Japanese, utterances depend on a **style** parameter, with values “plain” and “respect”.

Utterances are formed from different categories, for instance,

```
fun UttS : S -> Utt
lin UttS s = {s = s.s ! Dir}
```

```
fun UttQS : QS -> Utt
lin UttQS qs = {s = qs.s ! DirQuest}

fun UttNP : NP -> Utt
lin UttNP np = {s = np.s ! Nom}

fun UttAdv : Adv -> Utt
lin UttAdv adv = adv
```

The string that is used as linearization is selected by some typical parameter. For instance, sentences and questions use the "direct" instead of indirect or subordinate form, and noun phrases the nominative. Depending on language, more alternatives are needed in extensions of the core RGL — for instance, all case forms of NP that might be used as utterances that answer a question.

Of course, "direct" may imply a different word order for declaratives and questions. A typical rule in many languages is that a direct question has the order of an inverted declarative, and an indirect question has the order of a direct declarative:

Direct declarative: *she is here.*

Direct question: *is she here.*

Indirect question: (*I don't know*) *if she is here.*

6.2.2 Sentences and clauses

The sentence category is divided into two levels:

- **sentence** (S) has fixed **temporal features** (Temp, including **tense** and **aspect**) and **polarity** (Pol , positive or negative)
- **clause** (Cl), which has variable tense, aspect, and polarity

Sentences themselves can still have different forms for main clause and subordinate use, as well as questions. This can be expressed by an **order** parameter.

```
cat S
lincat S = {s : Order => Str}

cat Temp ; Pol
lincat Temp = {s : Str ; t : TenseForm}
lincat Pol  = {s : Str ; b : Bool}

cat Cl
lincat Cl = {
  subj : Str ;
  agr  : Agr ;
  verb : Verb ;
  comp : Str
  }

fun UseCl : Temp -> Pol -> Cl -> S  # aux advmod head
lin UseCl temp pol cl = {
  s = \\d => arrange d
               cl.subj
               (verbForm cl.verb cl.agr temp pol)
               cl.comp
  }

oper arrange  : Order -> Str -> Str^n -> Str -> Str
oper verbForm : Verb -> Agr -> Temp -> Pol -> Str^n
```

What is happening here? The clause is a vastly discontinuous constituents, where

- the subject, verb, and complement are put into desired order by the auxiliary `arrange` operation
- the verb contains the minimal inflection table (as in Section 5.3), from which different tenses and polarities are formed by the `verbForm` operation in agreement to the subject

The `verbForm` operator can produce tuples (or records) of strings, marked above as `Str^n`. For instance, in English, it produces a triple containing the infinite and finite verb forms and a negation word (empty string for positive polarity):

`verbForm run_V ASgP3 TPresent PNeg` $\Downarrow$ `<"does","not","run">`

The `arrange` operator shuffles them in different ways depending on order:

main clause: *she does not run*

question: *does she not run*

question, contracted negation: *does + n't she run*

Getting these details right has been the most challenging part of the English RGL, because of the auxiliary *do* in questions and negations. In other languages, `UseCl` can be relatively straightforward, since questions are formed by inversion and the negation is just a word inserted somewhere; the exact place of the negation can be a problem, however. We will return to the details of this in some languages in Chapter 7.

Clauses are formed from an NP and VP by the **predication** rule. To make sense of it, we need to define the linearization types of NP and VP here.

```
cat VP
lincat VP = {verb : Verb ; comp : Agr => Str}

cat NP
lincat NP = {s : Case => Str ; a : Agr}

fun PredVP : NP -> VP -> Cl  # nsubj head
lin PredVP np vp = {
  subj = np.s ! Nom ;
  agr  = np.a ;
  verb = vp.verb ;
  comp = vp.comp ! np.a
  }
```

One variation of this is verbs in e.g. Finnish and German that have different subject cases as an inherent feature. The `comp` part of the VP may agree to the subject; in English, this happens for instance when it contains a reflexive pronoun. The `comp` part is often divided into several parts, as for instance in the **topological structures** of Germanic languages or the control of clitics and other kinds of objects in Romance languages (see Section 7).

Many of the actual implementations in the RGL differ from the above schemas by shifting the load of `arrange` and `verbForm` one level lower and making the clause and the VP into tables instead of records:

```
lincat Cl = {s : Order => TenseForm => Bool => Str}
lincat VP = {s : Order => TenseForm => Bool => Agr => Str}
```

This set-up is somewhat easier to understand conceptually, but it does not scale up equally well to variant word orders. It can also become computationally heavy, because the tables that are formed can get huge: the VP could easily grow into a table of 1152 forms (with 4 orders, 8 tenses, 2 polarities, and the 18 agreement feature combinations of 3 genders, 2 numbers, and 3 persons).

Whatever the variations in the concrete syntax, we have now covered all of the NP-VP rules of Section 4.3 by just two abstract syntax functions: `UseCl` and `PredVP`. This includes the rules that use a copula and its complement, since they can be made into VPs, as we will see in Section 6.2.4.

To complete the picture of sentence formation (except for coordination), we need to deal with the case where the subject is an embedded sentence, in the category SC:

```
cat SC
lincat SC = {s : Str}

EmbedS : S -> SC
EmbedS s = {s = s.s ! Sub}

PredSCVP : SC -> VP -> Cl # csubj head
PredSCVP sc vp = PredVP (lin NP {s = \\_ => sc.s ; a = ASgP3}) vp
```

The last linearization rule forms an *ad hoc* noun phrase from the SC sentence by building a record of expected type. This record is wrapped with the `lin NP` operator to guide the type checker to treat it as an NP, even though it is not formed by regular NP-forming abstract syntax rules.

6.2.3 Verb phrases with verb heads

In contrast to the BNF rules in Section 4.3.3, we will now form verb phrases separately for different subcategories of verbs, as described in Section 5.5. For verbs that take NP complements, the formation is done in two phases: by first forming a VPSlash, and then adding the complement to it; this leaves room for Wh complements when questions and relative clauses are formed.

The VPSlash category itself is just like VP with a complement case (usually with a preposition included), in analogy to the way V2 is formed from V. But it also needs to keep track of the place of the new complement in relation to a possible old one. One way to do this is to introduce another complement, which normally occurs after the first complement. These two complement fields create a "hole" in which the new complement is inserted. The VPSlash also has to remember whether the agreement comes from the inserted object, as with object-control V2V verbs (*I order you to wash yourself*), or from the subject (*I want to wash myself*). This is done with the boolean `oc` feature.

```
cat VPSlash
lincat VPSlash = VP ** {
  comp2 : Agr => Str ;
  c : ComplCase ;
  oc : Bool ;
  }

fun ComplSlash : VPSlash -> NP -> VP  # head obj
lin ComplSlash vps np = vps ** {
  comp = \\a =>
    let agr = case vps.oc of {
      True => np.a ;
      False => a
      }
    in
      vps.comp ! agr ++
      complForm np vps.c ++
      vps.comp2 ! agr
  }

oper complForm : NP -> ComplCase -> Str
```

Notice the use of record extension in `ComplSlash`: it retains all VPSlash fields as they are, except that

- the new NP complement is added to the old complement
- the compiler will automatically delete the `c` field, because it is not a part of the linearization type of VP

The operation `compForm` selects the proper case of the NP and adds the preposition if there is one.

We should also notice that the UD label for the complement is `obj`, even if the object has a preposition, in which case UD would choose `obl`. The choice between `obl` and `obj` is not possible at the abstract syntax level, and it is not uniform across languages. Therefore one might question whether it is actually a good decision of UD to make this distinction at all. However, if we want to be pragmatic and support for instance generation of synthetic UD treebanks, we can refine the labelling by using language-dependent annotation rules, to be discussed in Section 6.3.

Now we are ready to list the VP formation rules corresponding to the subcategories of verbs:

```
fun UseV : V -> VP
lin UseV v =
  {verb = v ; comp = \\_ => []}

fun SlashV2 : V2 -> VPSlash
lin SlashV2 v = UseV v **
  {comp2 = \\_ => [] ; c = v.c ; oc = False}

fun Slash2V3 : V3 -> NP -> VPSlash  # head iobj
lin Slash2V3 v np = SlashV2 v **
  {comp = \\_ => complForm np v.c1 ; c = v.c2}

fun Slash3V3 : V3 -> NP -> VPSlash  # head obj
lin Slash3V3 v np = SlashV2 v **
  {comp2 = \\_ => complForm np v.c2 ; c = v.c1}

fun ComplVS : VS -> S -> VP  # head ccomp
lin ComplVS v s = UseV v **
  {comp = \\_ => that ++ s.s ! Sub}

oper that : Str

fun ComplVV : VV -> VP -> VP  # head xcomp
lin ComplVV v vp = UseV v **
  {comp = \\a => vpForm vp v.i a}

fun SlashV2V : V2V -> VP -> VP  # head xcomp
lin SlashV2V v vp = UseV v **
  {comp2 = \\a => vpForm vp v.i a ; c = v.c ; oc = v.oc}
```

```
oper vpForm : VP -> InfForm -> Agr -> Str
```

Notice that we can use `UseV` in a uniform way to take care of the verb part of the VP, and just overwrite the empty complement. This is similar to the use of **prototypes** in object-oriented programming: a prototype VP has a verb and an empty complement. In the same way, `SlashV2` gives a prototype VPSlash (but its official RGL is `SlashV2a` for historical reasons).

The two versions of V3 slash formation correspond to whether the middle or the last argument is inserted first. We indicate the place of the corresponding non-wh argument with the symbol Ø:

```
SlashV2V3 give_V3 she_NP
{comp = \\_ => "her" ; comp2 = \\_ => []}
```
what did you give her Ø

```
SlashV3V3 give_V3 it_NP
{comp = \\_ => [] ; comp2 = \\_ => "it"}
```
whom did you give Ø it

The UD annotation uses this information when selecting `iobj` for the middle argument. However, the correct label could be `obl` as well. What is more, in the latter example, the label given to *whom* is `obj` in accordance with the `ComplSlash` rule. But the semantically relevant information about which object does not get lost, as it is encoded in the abstract syntax tree.

Verb phrases can be modified by "ordinary" adverbials (Adv) and sentence adverbials (AdV in the RGL).

```
fun AdvVP : VP -> Adv -> VP # head advmod
lin AdvVP vp adv = vp ** {comp = \\a => vp.comp ! a ++ adv.s}

fun AdVVP : AdV -> VP -> VP # advmod head
lin AdVVP adv vp = vp ** {preverb = adv.s}
```

The latter rule requires in many languages a new field in the VP, `preverb`, to store material that appears between the subject and the verb. The order of the arguments in the abstract syntax functions is of course just mnemonic, and precludes in no way the linear position of the adverb in different languages.

What about passive verb phrases? In English, they can be built from the copula and a passive participle form of a VPSlash:

```
fun PassVPSlash : VPSlash -> VP
lin PassVPSlash vps = UseV copula **
  {comp = \\a => vpForm vps passPart a}
```

Notice that the result is an ordinary VP, which gives no information to select the UD label `nsubj:pass` when the VP is combined with a subject. (The Core RGL has a more specific function `PassV2`, but many languages implement this more general function in extension modules.)

Another way of converting VPSlash to VP without any supplementary arguments is by **reflexivization**: by filling the argument place with a **reflexive pronoun** (*myself*, *yourself*, etc). The reflexive pronoun will eventually be chosen in agreement to the subject, which is why the `comp` part of the VP depends on the agreement features of the NP even in languages like English:

```
fun ReflVP : VPSlash -> VP
lin ReflVP vps = vps ** {
  comp = \\a =>
    let agr = case vps.oc of {
      True => np.a ;
      False => a
      }
    in
      vps.comp ! agr ++
      reflPron agr vps.c ++
      vps.comp2 ! agr
  }

oper reflPron : Agr -> ComplCase -> Str
```

Notice that the reflexive pronoun is syncategorematic, and its UD label must be assigned in concrete syntax annotions. In some languages, it might not be a separate word at all, but a part of the verb inflection.

6.2.4 Complements of the copula

Verb phrases may be built from adjectives, noun phrases, common nouns, and adverbials together with a copuls. The copula is however not a part of the abstract syntax but introduced in linearization.

```
cat Comp
lincat Comp = {s : Agr => Str}

fun UseComp : Comp -> VP
lin UseComp cmp = UseV copula ** {comp = cmp}

oper copula : Verb

fun CompAP : AP -> Comp
lin CompAP ap = {s = \\a => agrForm ap a}

oper agrForm : AP -> Agr -> Str

fun CompAdv : Adv -> Comp
lin CompAdv adv = {s = \\_ => adv.s}

fun CompNP : NP -> Comp
lin CompNP np = {s = \\_ => np.s ! Nom}

fun CompCN : CN -> Comp
lin CompCN cn = {s = \\a => indefCN cn a Nom}

oper indefCN : CN -> Agr -> Str
```

Notice the difference between `CompNP` and `CompCN`: the latter agrees to the subject. Thus the sentences

> *I am a student*
> *we are students*

have the same complement formed by `CompCN` from the CN *student*, whereas

> *I am a disaster*
> *we are a disaster*

have the same complement formed by `CompNP` from the indefinite NP *a disaster*.

6.2.5 Noun phrases, adjectives, and adverbials

Noun phrases are typically inflected for case and have inherent agreement features. The simplest way to build them is by lexical insersion from proper names (PN) and personal pronouns (Pron). The linearization types are repetition from earlier sections:

```
cat NP ; Pron ; PN
lincat NP, Pron, PN = {s : Case => Str ; a : Agr}
lincat PN = {s : Case => Str ; a : {g : Gender ; n : Number}}

fun UsePron : Pron -> NP
lin UsePron pron = pron

fun UsePN : PN -> NP
lin UsePN pn = pn ** {a = pn.a ** {p = P3}}
```

Notice the definition of `Agr` in NP and PN as a record of gender, number and person. In PN, we do not need to include person, because it is always the third, automatically added by `UsePN`.

Another way to form noun phrases is by determiners from common nouns:

```
cat Det
lincat Det = {s : Gender => Case => Str ; n : Number}

cat CN
lincat CN = {s : Number => Case => Str ; g = Gender}

fun DetCN : Det -> CN -> NP
lin DetCN det cn = {
  s = \\c => det.s ! cn.g ! c ++ cn.s ! det.n ! c ;
  a = {g = cn.g ; n = det.n ; p = P3}
  }
```

Notice that the agreement features come from two sources: the number from the determiner, the gender from the common noun.

It is possible to build a noun phrases from nouns without determiners, as if there was an empty determiner. This makes typically sense only for **mass nouns** (such as *water*), including abstract nouns (*grammaticality*). But the RGL does not make the distinction on the level of syntax:

```
fun MassNP : CN -> NP
lin MassNP cn = DetCN (lin Det {s = \\_,_,_ => [] ; n = Sg}) cn
```

Also determiners without nouns can become NPs, as if with an empty CN.

```
fun DetNP : Det -> NP
lin DetNP det = DetCN det (lin CN {s = \\_,_ => [] ; g = Neutr})
```

For many determiners, this works with the same form of the determiner that is used in `DetCN`: *this*, *this cat*. But for some, depending on language, a specific **substantival form** is needed in the Det record: *mine*, *my cat*. A straightforward way to enable this is by adding another field `subst` to the linearization, with the same type as the normal `s` field:

```
lincat Det = {s,subst : Gender => Case => Str ; n : Number}
```

Notice that the `subst` field has gender variation in languages where, unlike English, determiners have gender-dependent forms (e.g. French *celui-ci*, *celle-ci*, "this" in its substantival form. For such languages, gender-dependent variants of `DetNP` are needed as abstract syntax functions. Such functions are, however, not completely interlingual: they are typically used as **anaphoric** expressions referring to some entities, so that for instance *this* often refers to the last-mentioned thing in the text. Just like anaphoric pronouns (*he, she, it*), the word *this* must then agree to the gender of the thing it refers to. The gender can vary across languages, so that for instance feminine *this* may have to be changed to masculine *this*. We will return to this problem in Section 8.4.

As we noticed in Section 5.5, many determiners vary in number, e.g. *this* - *these*. The RGL uses the category symbol Quant (**quantifier**) for them. Just like Det, Quant needs a record field for substantival uses:

```
cat Quant
lincat Quant = {s,subst : Number => Gender => Case => Str}
```

In the RGL syntax, Quant is made into a Det by combining it with a **number modifier**, Num:

```
cat Num
lincat Num = Det

fun DetQuant : Quant -> Num -> Det
lin DetQuant quant num = {
  s = \\g,c => quant.s ! num.n ! g ! c ++ num.s ! g ! c ;
  -- similar for subst
  n = num.n
  }
```

A Num can be built from a Numeral (*one, two, twenty-three*) or Digits (*1, 2, 23,000*), to be introduced in Section 6.2.10. The limiting case is empty numeral modifiers, which just specify the number:

```
fun NumSg, NumPl : Num
lin NumSg = {s,subst = \\_,_ => [] ; n = Sg}
lin NumPl = {s,subst = \\_,_ => [] ; n = Pl}
```

Thus we get the following analyses for *this cat, these cats*:

> `DetCN (DetQuant this_Quant NumSg) (UseN cat_N)`
> *this cat*
>
> `DetCN (DetQuant this_Quant NumPl) (UseN cat_N)`
> *these cats*

This slightly cumbersome analysis of determiner structure is the result of trying to compress the set of rules to a minimal set of orthogonal, binary rules. The same ambition has led to the treatment of indefinite and definite articles as quantifiers:

```
fun IndefArt, DefArt : Quant
```

In many languages, the strings are empty (for instance, the plural form of the indefinite article). The substantival forms often come out a bit artificial, for instance, as demonstrative pronouns (*it*) for the definite article. The treatment of **possessive pronouns** as quantifiers comes out a bit more natural, with e.g. *my, mine*:

```
fun PossPron : Pron -> Quant
```

To finish the discussion of determiner structure, we introduce **predeterminers**, which can be applied to already formed noun phrases that may contain their own determiners (e.g. ***only*** *these cats*, ***all*** *this beer*):

```
cat Predet
lincat Predet = {s : Gender => Number => Case => Str}

fun PredetNP : Predet -> NP -> NP
lin PredetNP predet np = {
  s = \\c => predet.s ! np.a.g ! np.a.n ! c ++ np.s ! c ;
  a = np.a
  }
```

Notice that *all*, which "logically" is a quantifier, syntactically works as a predeterminer. Thus the analysis of *all cats* is a bit cumbersome, involving the (empty) plural indefinite article:

```
PredetNP all_Predet
    (DetCN (DetQuant IndefArt NumPl) (UseN cat_N))
```

all cats

6.2.6 Common nouns, adjectives, and adverbials

Common nouns

```
fun UseN : N -> CN
lin UseN n = n

cat AP
lincat AP = {
  s : Gender => Number => Case => Str ;
  isPre : Bool
  }

fun AdjCN : AP -> CN -> CN
lin AdjCN ap cn = {
  s = \\c,n => case ap.isPre of {
    True  => ap.s ! cn.g ! n ! c ++ cn.s ! n ! c ;
```

```
      False => cn.s ! n ! c ++ ap.s ! cn.g ! n ! c ;
      } ;
    g = cn.g
    }

  fun AdvCN : CN -> Adv -> CN
  lin AdvCN cn adv = {
    s = \\c,n => cn.s ! n ! c  ++ adv.s ;
    g = cn.g
    }

  fun RelCN : RS -> CN -> CN
  lin RelCN rs cn = {
    s = \\n,c =>
      cn.s ! n ! c  ++ rs.s ! {g = cn.g ; n = n ; p = P3} ;
    g = cn.g
    }

  fun RelNP : RS -> NP -> NP
  lin RelNP rs np = {
    s = \\c => np.s ! c ++ comma ++ rs.s ! np.a ;
    a = np.a
    }

  fun SentCN : CN -> SC -> CN
  lin SentCN cn sc = {
    s = \\c,n => cn.s ! n ! c  ++ sc.s ;
    g = cn.g
    }

  fun ApposCN : CN -> NP -> CN
```

6.2.7 Adjectives, and adverbials

Adjectival phrases

```
  fun PositA : A -> AP
  lin PositA a = {
    s = a.s ! Posit ;
    isPre = True
    }

  fun ComparA : A -> NP -> AP
  lin ComparA a np = {
    s = \\g,n,c => a.s ! Compar ! g ! n ! c ++ complForm np than ;
    isPre = False
    }

  fun UseComparA : A -> AP
  lin UseComparA a  = {
    s = \\g,n,c => a.s ! Compar ! g ! n ! c ;
    isPre = True
    }


  fun AdAP : AdA -> AP -> AP
  lin AdAP ada ap = ap ** {
    s = \\g,n,c => ada.s ++ ap.s ! g ! n ! c ;
    }

  fun AdvAP : AP -> Adv -> AP
  lin AdvAP ap adv = {
    s = \\g,n,c => ap.s ! g ! n ! c ++ adv.s ;
    isPre = False
    }
```

Adverbial phrases

```
  fun PositAdvAdj : A -> Adv
  lin PositAdvAdj a = {s = a.adv ! Posit}

  cat Prep
  lincat Prep = ComplCase
```

```
fun PrepNP : Prep -> NP -> Adv
lin PrepNP prep np = complForm np prep

fun AdAdv : AdA -> Adv -> Adv
lin AdAdv ada adv = {s = ada.s ++ adv.s}

fun SubjS : Subj -> S -> Adv
lin SubjS subj s = {s = subj.s ++ s.s ! Sub}
```

6.2.8 Questions, relatives, and imperatives

There are two main ways to form questions:

- sentential questions (yes/no)
- wh questions, with an interrogative phrase (IP, *who*, *which* CN) or adverbial (IAdv, *why*)

Compared with the BNF rules in Section 4.3.5, these ways can be covered with four abstract syntax functions, which are moreover precise (not overgenerating). In addition, there is a conversion of **question clauses** (QCl) to **question sentences** (QS) similar to the conversion from Cl to S, by fixing temporal features and polarity.

```
cat QS
lincat QS = {s : QuestOrder => Str}

cat QCl
lincat QCl = Cl ** {q : Str}

cat IP
lincat IP = NP

fun UseQCl : Temp -> Pol -> QCl -> QS # aux advmod head
lin UseQCl t p qcl = {
  s = \\d => arrangeQuestion
          d qcl.q qcl.subj
 (verbForm qcl.verb qcl.agr temp pol)
 qcl.comp
  }
  fun QuestCl : Cl -> QCl
  lin QuestCl cl = cl ** {q = []}

  fun QuestVP : IP -> VP -> QCl # nsubj head
  lin QuestVP ip vp = PredVP ip vp ** {q = []}
```

Here we have assumed that the linearization types for questions match those for declaratives, only extending them with a field `q` for a Wh phrase to be added before the subject. This assumption can in fact be maintained if the clause types (QCl and Cl) are discontinuous enough. The auxiliary operation `arrangeQuestion` is similar to `arrange` in Section 6.2.2 except that it also places the interrogative element `qcl.q` in its proper place.

`QuestVP` places the interrogative on the subject position. To put it on another position, we introduce a category ClSlash, which is formed from VPSlash by adding a subject:

```
  cat ClSlash
  lincat ClSlash = {
    subj : Str ;
    verb : Verb ;
    comp : Agr => Str ;
    c : ComplCase
    }

  fun SlashVP : NP -> VPSlash -> ClSlash # nsubj head
  lin SlashVP np vps = PredVP np vps ** {
    comp = \\a =>
        let agr = case vps.oc of {
              True => a ;
      False => np.a
      }
in vps.comp ! agr ++ vps.comp2 ! agr ;
    c = vps.c
    }
```

Notice the subtle dependence of the complement on agreement: if the VP-Slash is object-control, the decision must be postponed,

which boy did you order to wash himself

which girl did you order to wash herself

But if it is subject-control, the complement is fixed when SlashVP is applied:

> *to which boy did you promise to wash yourself*
>
> *to which girl did you promise to wash yourself*

Now we can finally define the formation of Wh questions for non-subject positions:

```
fun QuestSlash : IP -> ClSlash -> QCl
lin QuestSlash ip cls = cls ** {
  comp = cls.comp ! ip.a ;
  q = complForm ip cls.c
  }
```

This rule implements the so-called **pied-piping** version of non-subject Wh phrases, where the preposition is attached to the Wh phrase (*to which girl*). English and some other languages also have a **preposition stranding** variant, where the preposition is appended to the end of the sentence:

> *which girl did you promise to wash yourself to*

The corresponding rule is

```
fun StrandQuestSlash : IP -> ClSlash -> QCl
lin StrandQuestSlash ip cls = cls ** {
  comp = cls.comp ! ip.a ++ cls.c.s
  q = ip.s ! Acc
  }
```

where the "complement case" is assumed to include a preposition in the `s` field.

Question with interrogative adverbials are simple: just add the IAdv to the clause record,

```
fun QuestIAdv : IAdv -> Cl -> QCl
lin QuestIAdv iadv cl = cl ** {q = iadv.s}
```

Relative clauses do not need different orders, because they are used only in subordinate positions. However, they need to agree to their correlates:

(*the boy*) *who washed himself*

(*the girl*) *who washed herself*

Therefore we add an agreement parameter to RS (corresponding to S and QS). RCl is related to QCl, but has a more complex agreement behaviour:

```
cat RS
lincat RS = {s : Agr => Str}

cat RCl
lincat RCl = {
  r, subj : Agr => Str
  agr  : RAgr ;
  verb : Verb ;
  comp : Agr => Str ;
  }

fun UseRCl : Temp -> Pol -> RCl -> RS # aux advmod head
lin UseRCl t p rcl = arrangeRelative -- ....
```

What makes the agreement of relative pronouns subtle is that an RP may just mediate the agreement of the correlate:

(*boy*) ***who*** *washed himself*

(*girl*) ***who*** *washed herself*

or have itself an agreement that the correlate cannot change:

(*boy*) ***whose sister*** *washed herself*

(*girl*) ***whose brother*** *washed himself*

This is encoded in the parameter

```
param RAgr = RANone | RAAgr Agr
```

where *who* has the `RANone` value, *whose* CN a value inherited from the CN.

```
cat RP
lincat RP = {s : Agr => Case => Str ; a : RAgr}

fun IdRP : RP
lin IdR = {s = ... ; a = RANone}

fun FunRP : Prep -> NP -> RP -> RP # case nmod head
lin FunRP prep np rp = {
  s = \\a,c => np.s ! c ++ complForm {s = rp.s ! a ; a = a} prep ;
  a = RAAgr np.a
  }

fun GenRP : Num -> CN -> RP # nummod head
lin GenRP num cn = {
  s = \\a,c => IdRP.s ! a ! Gen ++ num.s ! cn.g ! c ++ cn.s ! num.n ! c ;
  a = RAAgr {g = cn.g ; n = num.n}
  }
```

This makes the rule for subject position relative pronouns somewhat complex:

```
fun RelVP : RP -> VP -> RCl
lin RelVP rp vp = {
  r    = \\ => [] ;
  subj = rp.s ! Nom ;
  agr  = rp.a ;
  verb = vp.verb ;
  comp = vp.comp
  }
```

For non-subject relatives, we have

```
fun RelSlash : RP -> ClSlash -> RCl
lin RelSlash rp cls = {
  r = complFormRP rp cls.c ;
  subj = \\_ => [] ;
  verb = cls.verb ;
  comp = \\a => cls.comp ! rp.a ; ----
  }
```

Imperatives

```
  cat Imp

  fun ImpVP : VP -> Imp
```

Using indirect questions

```
  cat VQ
  lincat VQ

  fun CompVQ : VQ -> QS -> VP

  fun EmbedQS : QS -> SC
```

6.2.9 Coordination

```
  cat ListX

  fun BaseX : X -> X -> X # head conj

  fun ConsX : X -> ListX -> X # head conj

  fun ConjX : Conj -> ListX -> X # cc head
```

6.2.10 Numerals and symbols

For the sake of curiosity, because so different from UD.

```
cat
  Digit ;       -- 2..9
  Sub10 ;       -- 1..9
  Sub100 ;      -- 1..99
  Sub1000 ;     -- 1..999
  Sub1000000 ;  -- 1..999999

fun
  num : Sub1000000 -> Numeral ; -- 123456
```

```
n2, n3, n4, n5, n6, n7, n8, n9 : Digit ; -- 2,3,4,5,6,7,8,9

pot01 : Sub10 ;                              -- 1
pot0 : Digit -> Sub10 ;                      -- d * 1
pot110 : Sub100 ;                            -- 10
pot111 : Sub100 ;                            -- 11
pot1to19 : Digit -> Sub100 ;                 -- 10 + d
pot0as1 : Sub10 -> Sub100 ;                  -- coerce 1..9
pot1 : Digit -> Sub100 ;                     -- d * 10
pot1plus : Digit -> Sub10 -> Sub100 ;        -- d * 10 + n
pot1as2 : Sub100 -> Sub1000 ;                -- coerce 1..99
pot2 : Sub10 -> Sub1000 ;                    -- m * 100
pot2plus : Sub10 -> Sub100 -> Sub1000 ;      -- m * 100 + n
pot2as3 : Sub1000 -> Sub1000000 ;            -- coerce 1..999
pot3 : Sub1000 -> Sub1000000 ;               -- m * 1000
pot3plus : Sub1000 -> Sub1000 -> Sub1000000 ; -- m * 1000 + n
```

Use of numerals:

```
cat Card ; Ord

fun NumNumeral : Numeral -> Card

fun OrdNumeral : Numeral -> Ord

fun NumCard : Card -> Num
```

May look strange:

```
fun OrdSuperl : A -> Ord

fun AdjOrd : Ord -> AP
```

Numerals from digits, to store agreement features:

```
cat Digits ; Dig

fun IDig  : Dig -> Digits ;           -- 8
fun IIDig : Dig -> Digits -> Digits ; -- 876

fun D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9 : Dig ;

fun NumDigits : Digits -> Card
```

Symbols and literals:

```
cat Symb
cat Int ; Float ; String

fun SymbPN  : Symb -> PN
fun IntPN   : Int -> PN
fun FloatPN : Float -> PN

fun MkSymb  : String -> Symb
```

6.3 From abstract syntax to dependencies

6.4 The Core Resource Grammar and its extensions

Chapter 7

A tour of languages (TO BE WRITTEN)

In this chapter, we will take a look at some RGL languages and the issues encountered in their concrete syntax. The first version is limited to languages where the author of the book has contributed, but this chapter will later be completed with contributions from other grammarians.

7.1 English

- auxiliary verbs, negation, questions
- the indefinite article *a/an*

7.2 German, Dutch, and Afrikaans

- the topological structure
- discontinuous NP and CN, non-projective dependency trees
- German subject case variation
- Afrikaans double negation

7.3 Scandinavian languages

- the topological structure
- definiteness and determiner types

- functor for mainland Scandinavian
- Icelandic separate

7.4 Romance languages and Latin

- functor for modern Western Romance
- Romanian and Latin separate
- the tense system
- the clitics system
- subjunctive mood

7.5 Slavic languages

- tense and aspect
- number agreement

7.6 Finnish and Estonian

- rich Morphology: tables vs. stemming
- expressions for definiteness
- the accusative case
- discourse clitics

7.7 Arabic and Maltese

- non-concatenative morphology
- tables vs. stemming

7.8 Bantu languages

- prefix classes and agreement
- working around missing adjectives

7.9 Chinese and Thai

- tokenization in the absence of word boundaries
- classifiers
- the place of adverbials
- reduplication
- phonetic and orthographic versions of the grammar

Chapter 8

Grammar and semantics (TO BE WRITTEN)

8.1 Abstract syntax as semantic hub

- the compiler tradition
- using Python dataclasses and pattern matching
- using Haskell datatypes
- almost compositional functions

8.2 Logical semantics

- semantic types of parts of speech
- Montague-style rules
- implementation in Python and Haskell

8.3 Semantic grammars and constructions

- the GDPR example

8.4 Semantic analysis beyond utterances

- context and anaphora
- grammars for texts and dialogues

Chapter 9

Grammar-based systems (TO BE WRITTEN)

9.1 Accessing GF from other languages

The focus will be on GF bindings in Python, which has become the most popular programming language in the natural language processing community. Similar techniques are available for Haskell and Java, but we refer to on-line documentation for details.

9.2 Translation

- the interlingual model

9.3 Multilingual generation

- almost compositional functions
- example: generation from logic
- example: Abstract Wikipedia

9.4 Text analysis

9.5 Interactive systems

Chapter 10

Algorithms for grammar-based language processing (TO BE WRITTEN)

The focus in this book has been on describing grammars and implementing them with available tools, in particular GF. The actual processing — parsing, linearization, translation — is taken care of tools such as GF and dependency parsers. The computational grammarian's need to know them are similar to any programmer's: they can write programs in a high level language without knowing the compiler or run-time system of the language.

This said, it is always interesting to know how things work under the hood. This knowledge can also have practical value, since it can guide grammarians to choose coding alternatives that result in optimal processing.

10.1 Morphological analysis and generation

10.2 Part of speech tagging

10.3 Context-free parsing

10.4 Dependency parsing

10.5 Parallel context-free parsing

10.6 Statistical disambiguation

10.7 Semantic disambiguation

10.8 Hybrid systems

10.9 Evaluation

10.9.1 Machine translation evaluation

10.9.2 Dependency parser evaluation

The evaluation metrics measures the agreement between dependency trees: what percentage of words have been labelled correctly. There are two variants:

- **Labelled Attachment Score** (LAS): "correctly" means that both the head (i.e.\ the position number of the head) and the label to be tested are equal to the head and label in the gold standard.
- **Unlabelled Attachment Score** (UAS): "correctly" means only that the head to be tested is equal to the head in the gold standard.

The UAS score is obviously always at least as high as LAS.

This metrics can be assigned to every dependency tree individually. When computed for a set of trees, there are two options:

- **Micro-average**: the average score for all words in the set:

$$MIC = \frac{\textit{\#correct-words}}{\textit{\#words-in-total}}$$

- **Macro-average**: the average of the scores for each tree in the set.

$$MAC = \frac{\Sigma \textit{\#MIC-per-sentence}}{\textit{\#sentences}}$$

The outcomes can be different as soon as the set contains trees of different sizes. For instance, if we have two trees
- *t1* of size 5, 4 correct, 1 wrong
- *t2* of size 15, 9 correct, 6 wrong

then
- $MIC = (4 + 9)/(5 + 15) = 13/20 = 0.65$
- $MAC = (0.8 + 0.6)/2 = 1.4/20 = 0.7$

We leave it as an exercise to find a situation where the macro average is lower than micro.

References (TO BE CREATED)