Excerpts from: Learn Prolog Now!

by
Patrick Blackburn
Johan Bos
Kristina Striegnitz

patrick@aplog.org
jbos@cogsci.ed.ac.uk
kris@coli.uni-sb.de

The full book is available online:

http://www.coli.uni-sb.de/~kris/learn-prolog-now

Contents

1

Facts, Rules, and Queries

This introductory lecture has two main goals:

1. To give some simple examples of Prolog programs. This will introduce us to the three basic constructs in Prolog: **facts**, **rules**, and **queries**. It will also introduce us to a number of other themes, like the role of **logic** in Prolog, and the idea of performing **matching** with the aid of **variables**.

2. To begin the systematic study of Prolog by defining **terms**, **atoms**, **variables** and other syntactic concepts.

1.1 Some simple examples

There are only three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a **knowledge base** (or a **database**) and Prolog programming is all about writing knowledge bases. That is, Prolog programs simply *are* knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting. So how do we *use* a Prolog program? By posing queries. That is, by asking questions about the information stored in the knowledge base. Now this probably sounds rather strange. It's certainly not obvious that it has much to do with programming at all – after all, isn't programming all about telling the computer what to do? But as we shall see, the Prolog way of programming makes a lot of sense, at least for certain kinds of applications (computational linguistics being one of the most important examples). But instead of saying more about Prolog in general terms, let's jump right in and start writing some simple knowledge bases; this is not just the *best* way of learning Prolog, it's the *only* way ...

1.1.1 Knowledge Base 1

Knowledge Base 1 (KB1) is simply a collection of facts. Facts are used to state things that are unconditionally true of the domain of interest. For example, we can state that Mia, Jody, and Yolanda are women, and that Jody plays air guitar, using the following four facts:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
```

This collection of facts is KB1. It is our first example of a Prolog program. Note that the names `mia`, `jody`, and `yolanda`, and the properties `woman` and `playsAirGuitar`, have been written so that the first letter is in lower-case. This is important; we will see why a little later.

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

```
?- woman(mia).
```

Prolog will answer `yes` for the obvious reason that this is one of the facts explicitly recorded in KB1. Incidentally, *we* don't type in the `?-`. This symbol (or something like it, depending on the implementation of Prolog you are using) is the prompt symbol that the Prolog interpreter displays when it is waiting to evaluate a query. We just type in the actual query (for example `woman(mia)`) followed by `.` (a full stop).

Similarly, we can ask whether Jody plays air guitar by posing the following query:

```
?- playsAirGuitar(jody).
```

Prolog will again answer "yes", because this is one of the facts in KB1. However, suppose we ask whether Mia plays air guitar:

```
?- playsAirGuitar(mia).
```

We will get the answer `no`. Why? Well, first of all, this is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the *rules* we will learn about shortly) which might help Prolog try to **infer** (that is, **deduce** whether Mia plays air guitar. So Prolog correctly concludes that `playsAirGuitar(mia)` does *not* follow from KB1.

Here are two important examples. Suppose we pose the query:

```
?- playsAirGuitar(vincent).
```

Again Prolog answers "no". Why? Well, this query is about a person (Vincent) that it has no information about, so it concludes that `playsAirGuitar(vincent)` cannot be deduced from the information in KB1.

Similarly, suppose we pose the query:

```
?- tatooed(jody).
```

Again Prolog will answer "no". Why? Well, this query is about a property (being tatooed) that it has no information about, so once again it concludes that the query cannot be deduced from the information in KB1.

1.1.2 Knowledge Base 2

Here is KB2, our second knowledge base:

```
listensToMusic(mia).
happy(yolanda).
playsAirGuitar(mia)  :- listensToMusic(mia).
playsAirGuitar(yolanda) :- listensToMusic(yolanda).
listensToMusic(yolanda):- happy(yolanda).
```

KB2 contains two facts, `listensToMusic(mia)` and `happy(yolanda)`. The last three items are rules.

Rules state information that is *conditionally* true of the domain of interest. For example, the first rule says that Mia plays air guitar *if* she listens to music, and the last rule says that Yolanda listens to music *if* she is happy. More generally, the `:-` should be read as "if", or "is implied by". The part on the left hand side of the `:-` is called the **head** of the rule, the part on the right hand side is called the **body**. So in general rules say: *if* the body of the rule is true, *then* the head of the rule is true too. And now for the key point: *if a knowledge base contains a rule* `head :- body,` *and Prolog knows that* `body` *follows from the information in the knowledge base, then Prolog can infer* `head.` This fundamental deduction step is what logicians call **modus ponens**.

Let's consider an example. We will ask Prolog whether Mia plays air guitar:

```
?- playsAirGuitar(mia).
```

Prolog will respond "yes". Why? Well, although `playsAirGuitar(mia)` is not a fact explicitly recorded in KB2, KB2 does contain the rule

```
playsAirGuitar(mia) :- listensToMusic(mia).
```

Moreover, KB2 also contains the fact `listensToMusic(mia)`. Hence Prolog can use modus ponens to deduce that `playsAirGuitar(mia)`.

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask:

```
?- playsAirGuitar(yolanda).
```

Prolog would respond "yes". Why? Well, using the fact `happy(yolanda)` and the rule

```
listensToMusic(yolanda):- happy(yolanda),
```

Prolog can deduce the new fact `listensToMusic(yolanda)`. This new fact is not explicitly recorded in the knowledge base — it is only *implicitly* present (it is *inferred* knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. Thus, together with the rule

```
playsAirGuitar(yolanda) :- listensToMusic(yolanda)
```

it can deduce that `playsAirGuitar(yolanda)`, which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

The facts and rules contained in a knowledge base are called **clauses**. Thus KB2 contains five clauses, namely three rules and two facts. Another way of looking at KB2 is to say that it consists of three **predicates** (or **procedures**). The three predicates are:

```
listensToMusic
happy
playsAirGuitar
```

The `happy` predicate is defined using a single clause (a fact). The `listensToMusic` and `playsAirGuitar` predicates are each defined using two clauses (in both cases, two rules). It is a good idea to think about Prolog programs in terms of the predicates they contain. In essence, the predicates are the concepts we find important, and the various clauses we write down concerning them are our attempts to pin down what they mean and how they are inter-related.

One final remark. We can view a fact as a rule with an empty body. That is, we can think of facts as "conditionals that do not have any antecedent conditions", or "degenerate rules".

1.1.3 Knowledge Base 3

KB3, our third knowledge base, consists of five clauses:

```
happy(vincent).
listensToMusic(butch).
playsAirGuitar(vincent):-
   listensToMusic(vincent),
   happy(vincent).
playsAirGuitar(butch):-
   happy(butch).
playsAirGuitar(butch):-
   listensToMusic(butch).
```

There are two facts, namely `happy(vincent)` and `listensToMusic(butch)`, and three rules.

KB3 defines the same three predicates as KB2 (namely `happy`, `listensToMusic`, and `playsAirGuitar`) but it defines them differently. In particular, the three rules that define the `playsAirGuitar` predicate introduce some new ideas. First, note that the rule

```
playsAirGuitar(vincent):-
   listensToMusic(vincent),
   happy(vincent).
```

has *two* items in its body, or (to use the standard terminology) two **goals**. What does this rule mean? The important thing to note is the comma that separates the goal `listensToMusic(vincent)` and the goal `happy(vincent)` in the rule's body. This is the way logical **conjunction** is expressed in Prolog (that is, the comma means *and*). So this rule says: "Vincent plays air guitar if he listens to music and he is happy".

Thus, if we posed the query

```
?- playsAirGuitar(vincent).
```

Prolog would answer "no". This is because while KB3 contains `happy(vincent)`, it does *not* explicitly contain the information `listensToMusic(vincent)`, and this fact cannot be deduced either. So KB3 only fulfils one of the two preconditions needed to establish `playsAirGuitar(vincent)`, and our query fails.

Incidentally, the spacing used in this rule is irrelevant. For example, we could have written it as

```
playsAirGuitar(vincent):- happy(vincent),listensToMusic(vincent).
```

and it would have meant exactly the same thing. Prolog offers us a lot of freedom in the way we set out knowledge bases, and we can take advantage of this to keep our code readable.

Next, note that KB3 contains two rules with *exactly* the same head, namely:

```
playsAirGuitar(butch):-
   happy(butch).
playsAirGuitar(butch):-
   listensToMusic(butch).
```

This is a way of stating that Butch plays air guitar if *either* he listens to music, *or* if he is happy. That is, listing multiple rules with the same head is a way of expressing logical **disjunction** (that is, it is a way of saying *or*). So if we posed the query

```
?- playsAirGuitar(butch).
```

Prolog would answer "yes". For although the first of these rules will not help (KB3 does not allow Prolog to conclude that `happy(butch)`), KB3 *does* contain `listensToMusic(butch)` and this means Prolog can apply modus ponens using the rule

```
playsAirGuitar(butch):-
   listensToMusic(butch).
```

to conclude that `playsAirGuitar(butch)`.

There is another way of expressing disjunction in Prolog. We could replace the pair of rules given above by the single rule

```
playsAirGuitar(butch):-
   happy(butch);
   listensToMusic(butch).
```

That is, the semicolon `;` is the Prolog symbol for *or*, so this single rule means exactly the same thing as the previous pair of rules. But Prolog programmers usually write multiple rules, as extensive use of semicolon can make Prolog code hard to read.

It should now be clear that Prolog has something do with logic: after all, "`:-`" means implication, "`,`" means conjunction, and "`;`" means disjunction. (What about negation? That is a whole other story. We'll be discussing it later in the course.) Moreover, we have seen that a standard logical proof rule (modus ponens) plays an important role in Prolog programming. And in fact "Prolog" is short for "Programming in logic".

1.1.4 Knowledge Base 4

Here is KB4, our fourth knowledge base:

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
```

Now, this is a pretty boring knowledge base. There are no rules, only a collection of facts. Ok, we are seeing a relation that has two names as arguments for the first time (namely the `loves` relation), but, let's face it, that's a rather predictable idea.

No, the novelty this time lies not in the knowledge base, it lies in the *queries* we are going to pose. In particular, *for the first time we're going to make use of variables*. Here's an example:

```
?- woman(X).
```

The `X` is a variable (in fact, any word beginning with an upper-case letter is a Prolog variable, which is why we had to be careful to use lower-case initial letters in our earlier examples). Now a variable isn't a name, rather it's a "placeholder" for information. That is, this query essentially asks Prolog: tell me which of the individuals you know about is a woman.

Prolog answers this query by working its way through KB4, from top to bottom, trying to **match** (or **unify**) the expression `woman(X)` with the information KB4 contains. Now the first item in the knowledge base is `woman(mia)`. So, Prolog matches `X` to `mia`, thus making the query agree perfectly with this first item. (Incidentally, there's a lot of different terminology for this process: we can also say that Prolog **instantiates** `X` to `mia`, or that it **binds** `X` to `mia`.) Prolog then reports back to us as follows:

```
X = mia
```

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn't just say "yes", it actually gave us the **variable binding**, or **instantiation** that led to success.

But that's not the end of the story. The whole point of variables — and not just in Prolog either — is that they can "stand for" or "match with" different things. And there is information about other women in the knowledge base. We can access this information by typing the following simple query

```
?-  ;
```

Remember that `;` means *or*, so this query means: *are there any more women*? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it matches `X` with `jody`, then the query agrees perfectly with the second entry in the knowledge base. So it responds:

```
X = jody
```

It's telling us that there is information about a second woman in KB4, and (once again) it actually gives us the value that led to success. And of course, if we press `;` a second time, Prolog returns the answer

```
X = yolanda
```

But what happens if we press `;` a *third* time? Prolog responds "no". No other matches are possible. There are no other facts starting with the symbol `woman`. The last four entries in the knowledge base concern the `love` relation, and there is no way that such entries can match a query of the form of the form `woman(x)`.

Let's try a more complicated query, namely

```
?- loves(marcellus,X),woman(X).
```

Now, remember that `,` means *and*, so this query says: *is there any individual* `X` *such that Marcellus loves* `X` *and* `X` *is a woman*? If you look at the knowledge base you'll see that there is: Mia is a woman (fact 1) and Marcellus loves Mia (fact 5). And in fact, Prolog is capable of working this out. That is, it can search through the knowledge base and work out that if it matches `X` with Mia, then both conjuncts of the query are satisfied (we'll learn in later lectures exactly how Prolog does this). So Prolog returns the answer `X = mia`.

This business of matching variables to information in the knowledge base is the heart of Prolog. For sure, Prolog has many other interesting aspects — but when you get right down to it, it's Prolog's ability to perform matching and return the values of the variable binding to us that is crucial.

1.1.5 Knowledge Base 5

Well, we've introduced variables, but so far we've only used them in queries. In fact, variables not only *can* be used in knowledge bases, it's only when we start to do so that we can write truly interesting programs. Here's a simple example, the knowledge base KB5:

```
loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).

jealous(X,Y) :- loves(X,Z),loves(Y,Z).
```

KB5 contains four facts about the `loves` relation and one rule. (Incidentally, the blank line between the facts and the rule has no meaning: it's simply there to increase the readability. As we said earlier, Prolog gives us a great deal of freedom in the way we format knowledge bases.) But this rule is by far the most interesting one we have seen so far: it contains three variables (note that `X`, `Y`, and `Z` are all upper-case letters). What does it say?

In effect, it is defining a concept of jealousy. It says that an individual `X` will be jealous of an individual `Y` if there is some individual `Z` that `X` loves, and `Y` loves that same individual `Z` too. (Ok, so jealously isn't as straightforward as this in the real world ...) The key thing to note is that this is a *general* statement: it is not stated in terms of `mia`, or `pumpkin`, or anyone in particular — it's a conditional statement about *everybody* in our little world.

Suppose we pose the query:

```
?- jealous(marcellus,W).
```

This query asks: can you find an individual `W` such that Marcellus is jealous of `W`? Vincent is such an individual. If you check the definition of jealousy, you'll see that Marcellus must be jealous of Vincent, because they both love the same woman, namely Mia. So Prolog will return the value `W = vincent`.

Now some questions for *you*, First, are there any other jealous people in KB5? Furthermore, suppose we wanted Prolog to tell us about all the jealous people: what query would we pose? Do any of the answers surprise you? Do any seem silly?

1.2 Prolog Syntax

Now that we've got some idea of what Prolog does, it's time to go back to the beginning and work through the details more carefully. Let's start by asking a very basic question: we've seen all kinds of expressions (for example `jody`, `playsAirGuitar(mia)`, and `X`) in our Prolog programs, but these have just been examples. Exactly what are facts, rules, and queries built out of?

The answer is **terms**, and there are four kinds of terms in Prolog: **atoms**, **numbers**, **variables**, and **complex terms** (or **structures**). Atoms and numbers are lumped together under the heading **constants**, and constants and variables together make up the **simple terms** of Prolog.

Let's take a closer look. To make things crystal clear, let's first get clear about the basic **characters** (or **symbols**) at our disposal. The *upper-case letters* are `A`, `B`, ..., `Z`; the *lower-case letters* are `a`, `b`, ..., `z`; the *digits* are `1`, `2`, ..., `9`; and the *special characters* are `+`, `-`, `*`, `/`, `<`, `>`, `=`, `:`, `.`, `&`, `~`, and `_`. The `_` character is called **underscore**. The blank *space* is also a character, but a rather unusual one, being invisible. A **string** is an unbroken sequence of characters.

1.2.1 Atoms

An atom is either:

1. A string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with a lower-case letter. For example: `butch`, `big_kahuna_burger`, and `m_monroe2.`

2. An arbitrary sequence of character enclosed in single quotes. For example '`Vincent`', '`The Gimp`', '`Five_Dollar_Shake`', '`&^%&#@$ &*`', and ' '. The character between the single quotes is called the **atom name**. Note that we are allowed to use spaces in such atoms — in fact, a common reason for using single quotes is so we can do precisely that.

3. A string of special characters. For example: `@=` and `====>` and `;` and `:-` are all atoms. As we have seen, some of these atoms, such as `;` and `:-` have a pre-defined meaning.

1.2.2 Numbers

Real numbers aren't particularly important in typical Prolog applications. So although most Prolog implementations do support **floating point numbers** or **floats** (that is, representations of real numbers such as 1657.3087 or π) we are not going to discuss them in this course.

But **integers** (that is: ... -2, -1, 0, 1, 2, 3, ...) are useful for such tasks as counting the elements of a list, and we'll discuss how to manipulate them in a later lecture. Their Prolog syntax is the obvious one: `23`, `1001`, `0`, `-365`, and so on.

1.2.3 Variables

A variable is a string of upper-case letters, lower-case letters, digits and underscore characters that starts *either* with an upper-case letter *or* with underscore. For example, `X`, `Y`, `Variable`, `_tag`, `X_526`, and `List`, `List24`, `_head`, `Tail`, `_input` and `Output` are all Prolog variables.

The variable _ (that is, a single underscore character) is rather special. It's called the *anonymous variable*, and we discuss it in a later lecture.

1.2.4 Complex terms

Constants, numbers, and variables are the building blocks: now we need to know how to fit them together to make complex terms. Recall that complex terms are often called structures.

Complex terms are build out of a **functor** followed by a sequence of **arguments**. The arguments are put in ordinary parentheses, separated by commas, and placed after the functor. The functor *must* be an atom. That is, variables *cannot* be used as functors. On the other hand, arguments can be any kind of term.

Now, we've already seen lots of examples of complex terms when we looked at KB1 – KB5. For example, `playsAirGuitar(jody)` is a complex term: its functor is `playsAirGuitar` and its argument is `jody`. Other examples are `loves(vincent,mia)` and, to give an example containing a variable, `jealous(marcellus,W)`.

But note that the definition allows far more complex terms than this. In fact, it allows us to to keep nesting complex terms inside complex terms indefinitely (that is, it is a *recursive definition*). For example

```
hide(X,father(father(father(butch))))
```

is a perfectly ok complex term. Its functor is `hide`, and it has two arguments: the variable `X`, and the complex term `father(father(father(butch)))`. This complex term has `father` as its functor, and another complex term, namely `father(father(butch))`, as its sole argument. And the argument of this complex term, namely `father(butch)`, is also complex. But then the nesting "bottoms out", for the argument here is the constant `butch`.

As we shall see, such nested (or recursively structured) terms enable us to represent many problems naturally. In fact the interplay between recursive term structure and variable matching is the source of much of Prolog's power.

The number of arguments that a complex term has is called its **arity**. For instance, `woman(mia)` is a complex term with arity 1, while `loves(vincent,mia)` is a complex term with arity 2.

Arity is important to Prolog. Prolog would be quite happy for us to define two predicates with the same functor but with a different number of arguments. For example, we are free to define a knowledge base that defines a two place predicate `love` (this might contain such facts as `love(vincent,mia)`), and also a three place `love` predicate (which might contain such facts as `love(vincent,marcellus,mia)`). However, if we did this, Prolog would treat the two place `love` and the three place `love` as completely different predicates.

When we need to talk about predicates and how we intend to use them (for example, in documentation) it is usual to use a suffix `/` followed by a number to indicate the predicate's arity. To return to KB2, instead of saying that it defines predicates

```
listensToMusic
happy
playsAirGuitar
```

we should really say that it defines predicates

```
listensToMusic/1
happy/1
playsAirGuitar/1
```

And Prolog can't get confused about a knowledge base containing the two different love predicates, for it regards the `love/2` predicate and the `love/3` predicate as completely distinct.

2

Matching and Proof Search

Today's lecture has two main goals:

1. To discuss the idea of **matching** in Prolog, and to explain how Prolog matching differs from standard **unification**. Along the way, we'll introduce `=`, the built-in Prolog predicate for matching.
2. To explain the search strategy Prolog uses when it tries to prove something.

2.1 Matching

When working with knowledge base KB4 in the previous chapter, we introduced the term *matching*. We said, e.g. that Prolog matches `woman(X)` with `woman(mia)`, thereby instantiating the variable `X` to `mia`. We will now have a close look at what *matching* means.

Recall that there are three types of term:

1. Constants. These can either be atoms (such as `vincent`) or numbers (such as `24`).
2. Variables.
3. Complex terms. These have the form: `functor(term_1,...,term_n)`.

We are now going to define when two terms match. The basic idea is this:

> Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.

That means that the terms `mia` and `mia` match, because they are the same atom. Similarly, the terms `42` and `42` match, because they are the same number, the terms `X` and `X` match, because they are the same variable, and the terms `woman(mia)` and `woman(mia)` match, because they are the same complex term. The terms `woman(mia)` and `woman(vincent)`, however, do not match, as they are not the same (and neither of them contains a variable that could be instantiated to make them the same).

Now, what about the terms `mia` and `X`? They are not the same. However, the variable `X` can be instantiated to `mia` which makes them equal. So, by the second part of the above definition, `mia` and `X` match. Similarly, the terms `woman(X)` and `woman(mia)` match, because they can be made equal by instantiating `X` to `mia`. How about `loves(vincent,X)` and `loves(X,mia)`? It is impossible to find an instantiation of `X` that makes the two terms equal, and therefore they don't match. Do you see why? Instantiating `X` to `vincent` would give us the terms `loves(vincent,vincent)` and `loves(vincent,mia)`, which are obviously not equal. However, instantiating `X` to mia, would yield the terms `loves(vincent,mia)` and `loves(mia,mia)`, which aren't equal either.

Usually, we are not only interested in the fact that two terms match, but we also want to know in what way the variables have to be instantiated to make them equal. And Prolog gives us this information. In fact, when Prolog matches two terms it performs all the necessary instantiations, so that the terms really are equal afterwards. This functionality together with the fact that we are allowed to build complex terms (that is, *recursively structured* terms) makes matching a quite powerful mechanism. And as we said in the previous chapter: matching is one of the fundamental ideas in Prolog.

Here's a more precise definition for matching which not only tells us *when* two terms match, but one which also tells us *what we have to do* to the variables to make the terms equal.

1. If `term1` and `term2` are constants, then `term1` and `term2` match if and only if they are the same atom, or the same number.

2. If `term1` is a variable and `term2` is any type of term, then `term1` and `term2` match, and `term1` is instantiated to `term2`. Similarly, if `term2` is a variable and `term1` is any type of term, then `term1` and `term2` match, and `term2` is instantiated to `term1`. (So if they are both variables, they're both instantiated to each other, and we say that they **share values**.)

3. If `term1` and `term2` are complex terms, then they match if and only if:

   (a) They have the same functor and arity.

   (b) All their corresponding arguments match

   (c) and the variable instantiations are compatible. (I.e. it is not possible to instantiate variable `X` to `mia`, when matching one pair of arguments, and to then instantiate `X` to `vincent`, when matching another pair of arguments.)

4. Two terms match if and only if it follows from the previous three clauses that they match.

Note the *form* of this definition. The first clause tells us when two constants match. The second term clause tells us when two terms, one of which is a variable, match: such terms will *always* match (variables match with *anything*). Just as importantly, this clause also tells what instantiations we have to perform to make the two terms the same. Finally, the third clause tells us when two complex terms match.

The fourth clause is also very important: it tells us that the first three clauses completely define when two terms match. If two terms can't be shown to match using Clauses 1-3, then they *don't* match. For example, `batman` does not match with `daughter(ink)`. Why not? Well, the first term is a constant, the second is a complex term. But none of the first three clauses tell us how to match two such terms, hence (by clause 4) they don't match.

2.1.1 Examples

We'll now look at lots of examples to make this definition clear. In these examples we'll make use of an important built-in Prolog predicate, the `=/2` predicate (recall that the `/2` at the end is to indicate that this predicate takes two arguments).

Quite simply, the `=/2` predicate tests whether its two arguments match. For example, if we pose the query

```
?- =(mia,mia).
```

Prolog will respond 'yes', and if we pose the query

```
?- =(mia,vincent).
```

Prolog will respond 'no'.

But we usually wouldn't pose these queries in quite this way. Let's face it, the notation `=(mia,mia)` is rather unnatural. It would be much nicer if we could use **infix** notation (that is, put the `=` functor *between* its arguments) and write things like: `mia = mia .` And in fact, Prolog lets us do this. So in the examples that follow we'll use the (much nicer) infix notation.

Let's return to this example:

```
?- mia = mia.
yes
```

Why does Prolog say 'yes'? This may seem like a silly question: surely it's obvious that the terms match! That's true, but how does this follow from the definition given above? It is very important that you learn to think systematically about matching (it is utterly fundamental to Prolog), and 'thinking systematically' means relating the examples to the definition of matching given above. So let's think this example through.

The definition has three clauses. Clause 2 is for when one argument is a variable, and clause 3 is for when both arguments are complex terms, so these are no use here. However clause 1 *is* relevant to our example. This tells us that two constants unify if and only if they are are exactly the same object. As `mia` and `mia` are the same atom, matching succeeds.

A similar argument explains the following responses:

```
?- 2 = 2.
yes

?- mia = vincent.
no
```

Once again, clause 1 is relevant here (after all, `2`, `mia`, and `vincent` are all constants). And as `2` is the same number as `2`, and as `mia` is *not* the same atom as `vincent`, Prolog responds 'yes' to the first query and 'no' to the second.

However clause 1 does hold one small surprise for us. Consider the following query:

```
?- 'mia' = mia.
yes
```

What's going here? Why do these two terms match? Well, as far as Prolog is concerned, `'mia'` and `mia` are the same atom. In fact, for Prolog, any atom of the form `'symbols'` is considered the same entity as the atom of the form `symbols`. This can be a useful feature in certain kinds of programs, so don't forget it.

On the other hand, to the the query

```
?- '2' = 2.
```

Prolog will respond 'no'. And if you think about the definitions given in Lecture 1, you will see that this has to be the way things work. After all, `2` is a number, but `'2'` is an atom. They simply cannot be the same.

Let's try an example with a variable:

```
?- mia = X.

X = mia
yes
```

Again, this in an easy example: clearly the variable `X` can be matched with the constant `mia`, and Prolog does so, and tells us that it has made this matching. Fine, but how does this follow from our definition?

The relevant clause here is clause 2. This tells us what happens when at least one of the arguments is a variable. In our example it is the second term which is the variable. The definition tells us unification is possible, and also says that the variable is instantiated to the first argument, namely `mia`. And this, of course, is exactly what Prolog does.

Now for an important example: what happens with the following query?

```
?- X = Y.
```

Well, depending on your Prolog implementation, you may just get back the output

```
?- X = Y.

yes
```

Prolog is simply agreeing that the two terms unify (after all, variables unify with anything, so certainly with each other) and making a note that from now on, `X` and `Y` denote the same object. That is, if ever `X` is instantiated, `Y` will be instantiated too, and to the same thing.

On the other hand, you may get the following output:

```
X = _5071
Y = _5071
```

Here, *both* arguments are variables. What does this mean?

Well, the first thing to realize is that the symbol `_5071` is a variable (recall from Lecture 1 that strings of letters and numbers that start with a `_` are variables). Now look at clause 2 of the definition. This tells us that when two variables are matched, they *share values*. So what Prolog is doing here is to create a new variable (namely `_5071`) and saying that, from now on, both `X` and `Y` share the value of this variable. That is, in effect, Prolog is creating a common variable name for the two original variables. Incidentally, there's nothing magic about the number `5071`. Prolog just needs to generate a brand new variable name, and

using numbers is a handy way to do this. It might just as well generate `_5075`, or `_6189`, or whatever.

Here is another example involving only atoms and variables. How do you think will Prolog respond?

```
?- X = mia, X = vincent.
```

Prolog will respond 'no'. This query involves two goals, `X = mia` and `X = vincent`. Taken seperately, Prolog would succeed for both of them, instantiating `X` to `mia` in the first case and to `vincent` in the second. And that's exactly the problem here: once Prolog has worked through the first query, `X` is instantiated, and therefore equal, to `mia`, so that that it doesn't match with `vincent` anymore and the second goal fails.

Now, let's look at an example involving complex terms:

```
?- kill(shoot(gun),Y) = kill(X,stab(knife)).

X = shoot(gun)
Y = stab(knife)
yes
```

Clearly the two complex terms match if the stated variable instantiations are carried out. But how does this follow from the definition? Well, first of all, Clause 3 has to be used here because we are trying to match two complex terms. So the first thing we need to do is check that both complex terms have the same functor (that is: they use the same atom as the functor name *and* have the same number of arguments). And they do. Clause 3 also tells us that we have to match the corresponding arguments in each complex term. So do the first arguments, `shoot(gun)` and `X`, match? By Clause 2, yes, and we instantiate `X` to `shoot(gun)`. So do the second arguments, `Y` and `stab(knife)`, match? Again by Clause 2, yes, and we instantiate `Y` to `stab(knife)`.

Here's another example with complex terms:

```
?- kill(shoot(gun), stab(knife)) = kill(X,stab(Y)).

X = shoot(gun)
Y = knife
yes
```

It should be clear that the two terms match if these instantiations are carried out. But can you explain, step by step, how this relates to the definition?

Here is a last example:

```
?- loves(X,X) = loves(marcellus,mia).
```

Do these terms match? No, they don't. They are both complex terms and have the same functor and arity. So, up to there it's ok. But then, Clause 3 of the definition says that all corresponding arguments have to match and that the variable instantiations have to be compatible, and that is not the case here. Matching the first arguments would instantiate `X` with `marcellus` and matching the second arguments would instantiate `X` with `mia`.

2.1.2 The occurs check

Instead of saying that Prolog matches terms, you'll find that many books say that Prolog **unifies** terms. This is very common terminology, and we will often use it ourselves. But while it does not really matter whether you call what Prolog does 'unification' or 'matching', there is one thing you *do* need to know: Prolog does not use a standard unification algorithm when it performs unification/matching. Instead, it takes a shortcut. You need to know about this shortcut.

Consider the following query:

```
?- father(X) = X.
```

Do you think these terms match or not? A standard unification algorithm would say: No, they don't. Do you see why? Pick any term and instantiate `X` to the term you picked. For example, if you instantiate `X` to `father(father(butch))`, the left hand side becomes `father(father(father(butch)))`, and the right hand side becomes `father(father(butch))`. Obviously these don't match. Moreover, it makes no difference what you instantiate `X` to. No matter what you choose, the two terms cannot possibly be made the same, for the term on the left will always be one symbol longer than the term on the right (the functor `father` on the left will always give it that one extra level). The two terms simply don't match.

But now, let's see what Prolog would answer to the above query. With old Prolog implementations you would get a message like:

```
Not enough memory to complete query!
```

and a long string of symbols like:

```
X = father(father(father(father(father(father(father(father
(father(father(father(father(father(father(father(father(father
(father(father(father(father(father(father(father(father(father
(father(father(father(father(father(father(father(father(father
(father(father(father(father(father(father(father(father(father
```

Prolog is desperately *trying* to match these terms, but it won't succeed. That strange variable `X`, which occurs as an argument to a functor on the left hand side, and on its own on the right hand side, makes matching impossible.

To be fair, what Prolog is trying to do here is reasonably intelligent. Intuitively, the only way the two terms could be made to match would be if `X` was instantiated to 'a term containing an infinitely long string of `father` functors', so that the effect of the extra `father` functor on the left hand side was canceled out. But terms are *finite* entities. There is no such thing as a 'term containing an infinitely long string of `father` functors'. Prolog's search for a suitable term is doomed to failure, and it learns this the hard way when it runs out of memory.

Now, current Prolog implementations have found a way of coping with this problem. Try to pose the query `father(X) = X` to SICStus Prolog or SWI. The answer will be something like:

```
X = father(father(father(father(father(father(...))))))))))
```

The dots are indicating that there is an infinite nesting of `father` functors. So, newer versions of Prolog can detect cycles in terms without running our of memory and have a finite internal representation of such infinite terms.

Still, a standard unification algorithm works differently. If we gave such an algorithm the same example, it would look at it and tell us that the two terms don't unify. How does it do this? By carrying out the **occurs check**. Standard unification algorithms always peek inside the structure of the terms they are asked to unify, looking for strange variables (like the *X* in our example) that would cause problems.

To put it another way, standard unification algorithms are *pessimistic*. They first look for strange variables (using the occurs check) and only when they are sure that the two terms are 'safe' do they go ahead and try and match them. So a standard unification algorithm will never get locked into a situation where it is endlessly trying to match two unmatchable terms.

Prolog, on the other hand, is *optimistic*. It assumes that you are not going to give it anything dangerous. So it does not make an occurs check. As soon as you give it two terms, it charges full steam ahead and tries to match them.

As Prolog is a programming language, this is an intelligent strategy. Matching is one of the fundamental processes that makes Prolog work, so it needs to be carried out as fast as possible. Carrying out an occurs check every time matching was called for would slow it down considerably. Pessimism is safe, but optimism is a lot faster!

Prolog can only run into problems if you, the programmer, ask it to do something impossible like unify `X` with `father(X)`. And it is unlikely you will ever ask it to anything like that when writing a real program.

2.1.3 Programming with matching

As we've said, matching is a fundamental operation in Prolog. It plays a key role in Prolog proof search (as we shall soon learn), and this alone makes it vital. However, as you get to know Prolog better, it will become clear that matching is interesting and important in its own right. Indeed, sometimes you can write useful programs simply by using complex terms to define interesting concepts. Matching can then be used to pull out the information you want.

Here's a simple example of this, due to Ivan Bratko. The following two line knowledge base defines two predicates, namely `vertical/2` and `horizontal/2`, which specify what it means for a line to be vertical or horizontal respectively.

```
vertical(line(point(X,Y),point(X,Z))).

horizontal(line(point(X,Y),point(Z,Y))).
```

Now, at first glance this knowledge base may seem too simple to be interesting: it contains just two facts, and no rules. But wait a minute: the two facts are expressed using complex terms which again have complex terms as arguments. If you look closely, you see that there are three levels of nesting terms into terms. Moreover, the deepest level arguments are all variables, so the concepts are being defined in a general way. Maybe its not quite as simple as it seems. Let's take a closer look.

Right down at the bottom level, we have a complex term with functor `point` and two arguments. Its two arguments are intended to be instantiated to numbers: `point(X,Y)` represents the Cartesian coordinates of a point. That is, the `X` indicates the horizontal distance the point is from some fixed point, while the `Y` indicates the vertical distance from that same fixed point.

Now, once we've specified two distinct points, we've specified a line, namely the line between them. In effect, the two complex terms representing points are bundled toghether as the two arguments of another complex term with the functor `line`. So, we represent a line by a complex term which has two arguments which are complex terms as well and represent points. *We're using Prolog's ability to build complex terms to work our way up a hierarchy of concepts*.

To be vertical or to be horizontal are properties of lines. The predicates `vertical` and `horizontal` therefore both take one argument which represents a line. The definition of `vertical/1` simply says: a line that goes between two points that have the same x-coordinate is vertical. Note how we capture the effect of 'the same x-coordinate' in Prolog: we simply make use of the same variable `X` as the first argument of the two complex terms representing the points.

Similarly, the definition of `horizontal/1` simply says: a line that goes between two points that have the same y-coordinate is horizontal. To capture the effect of 'the same y-coordinate', we use the same variable `Y` as the second argument of the two complex terms representing the points.

What can we do with this knowledge base? Let's look at some examples:

```
?- vertical(line(point(1,1),point(1,3))).
yes
```

This should be clear: the query matches with the definition of `vertical/1` in our little knowledge base (and in particular, the representations of the two points have the same first argument) so Prolog says 'yes'. Similarly we have:

```
?- vertical(line(point(1,1),point(3,2))).
no
```

This query does not match the definition of `vertical/1` (the representations of the two points have different first arguments) so Prolog says 'no'.

But we can ask more general questions:

```
?- horizontal(line(point(1,1),point(2,Y))).
Y = 1 ;
no
```

Here our query is: if we want a horizontal line between a point at (1,1), and point whose x-coordinate is 2, what should the y-coordinate of that second point be? Prolog correctly tells us that the y-coordinate should be 1. If we then ask Prolog for a second possibility (note the `;`) it tells us that no other possibilities exist.

Now consider the following:

```
?- horizontal(line(point(2,3),P)).
P = point(_1972,3) ;
no
```

This query is: if we want a horizontal line between a point at (2,3), and some other point, what other points are permissible? The answer is: any point whose y-coordinate is 3. Note that the `_1972` in the first argument of the answer is a variable, which is Prolog's way of telling us that any x-coordinate at all will do.

A general remark: the answer to our last query, `point(_1972,3)`, is *structured*. That is, the answer is a complex term, representing a sophisticated concept (namely 'any point whose y-coordinate is 3'). This structure was built using matching and nothing else: no logical inferences (and in particular, no uses of modus ponens) were used to produce it. Building structure by matching turns out to be a powerful idea in Prolog programming, far more powerful than this rather simple example might suggest. Moreover, when a program is written that makes heavy use of matching, it is likely to be extremely efficient. We will study a beautiful example in a later lecture when we discuss **difference lists**, which are used to implement Prolog's built-in grammar system **Definite Clause Grammars (DCGs)**.

This style of programming is particularly useful in applications where the important concepts have a natural hierarchical structure (as they did in the simple knowledge base above), for we can then use complex terms to represent this structure, and matching to access it. This way of working plays an important role in computational linguistics, because information about language has a natural hierarchical structure (think of the way we divide sentences into noun phrases and verb phrases, and noun phrases into determiners and nouns, and so on).

2.2 Proof Search

Now that we know about matching, we are in a position to learn how Prolog actually searches a knowledge base to see if a query is satisfied. That is, we are now able to learn about **proof search**. We will introduce the basic ideas involved by working through a simple example.

Suppose we are working with the following knowledge base

```
f(a).
f(b).

g(a).
g(b).

h(b).

k(X) :- f(X),g(X),h(X).
```

Suppose we then pose the query

```
?- k(X).
```

You will probably see that there is only one answer to this query, namely `k(b)`, but how exactly does Prolog work this out? Let's see.

Prolog reads the knowledge base, and tries to match `k(X)` with either a fact, or the head of a rule. It searches the knowledge base top to bottom, and carries out the matching, if it can, at the first place possible. Here there is only one possibility: it must match `k(X)` to the head of the rule `k(X) :- f(X),g(X),h(X)`.

When Prolog matches the variable in a query to a variable in a fact or rule, it generates a brand new variable to represent that the variables are now sharing. So the original query now reads:

```
k(_G348)
```

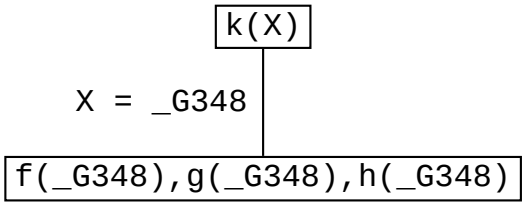
and Prolog knows that

```
k(_G348) :- f(_G348),g(_G348),h(_G348).
```

So what do we now have? The query says: 'I want to find an individual that has property `k`'. The rule says,'an individual has property `k` if it has properties `f`, `g`, and `h`'. So if Prolog can find an individual with properties `f`, `g`, and `h`, it will have satisfied the original query. So Prolog replaces the original query with the following list of goals:

```
f(_G348),g(_G348),h(_G348).
```

We will represent this graphically as

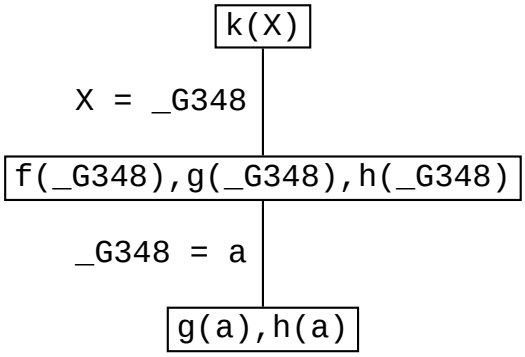


That is, our original goal is to prove `k(X)`. When matching it with the head of the rule in the knowledge base `X` and the internal variable `_G348` are made equal and we are left with the goals `f(_G348),g(_G348),h(_G348).`

Now, whenever it has a list of goals, Prolog tries to satisfy them one by one, working through the list in a left to right direction. The leftmost goal is `f(_G348)`, which reads: 'I want an individual with property `f`'. Can this goal be satisfied? Prolog tries to do so by searching through the knowledge base from top to bottom. The first thing it finds that matches this goal is the fact `f(a)`. This satisfies the goal `f(_G348)` and we are left with two more goals to go. When matching `f(_G348)` to `f(a)`, `X` is instantiated to `a`. This applies to all occurrences of `X` in the list of goals. So, the list of remaining goals is:

```
g(a),h(a)
```

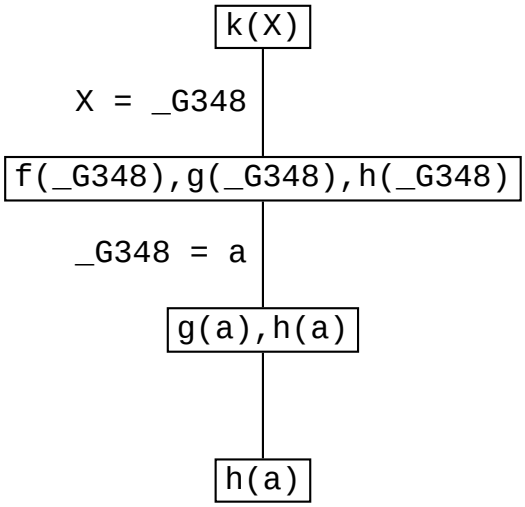
and our graphical representation of the proof search looks like this:



The fact `g(a)` is in the knowledge base. So the next goal we have to prove is satisfied too, and the goal list is now

```
h(a)
```

and the graphical representation



But there is no way to satisfy this goal. The only information `h` we have in the knowledge base is `h(b)` and this won't match `h(a)`.

So Prolog decides it has made a mistake and checks whether at some point there was another possibility for matching a goal with a fact or the head of a rule in the knowledge base. It

does this by going back up the path in the graphical representation that it was coming down on. There is nothing else in the knowledge base that matches with `g(a)`, but there is another possibility for matching `f(_G348)`. Points in the search where there are several alternatives for matching a goal against the knowledge base are called **choice points**. Prolog keeps track of choice points and the choices that it has made there, so that if it makes a wrong choice, it can go back to the choice point and try something else. This is called `backtracking`.

So, Prolog backtracks to the last choice point, where the list of goals was:

```
f(_G348),g(_G348),h(_G348).
```

Prolog has to redo all this. Prolog tries to **resatisfy** the first goal, by searching further in the knowledge base. It sees that it can match the first goal with information in the knowledge base by matching `f(_G348)` with `f(b)`. This satisfies the goal `f(_G348)` and instantiates `X` to `b`, so that the remaining goal list is

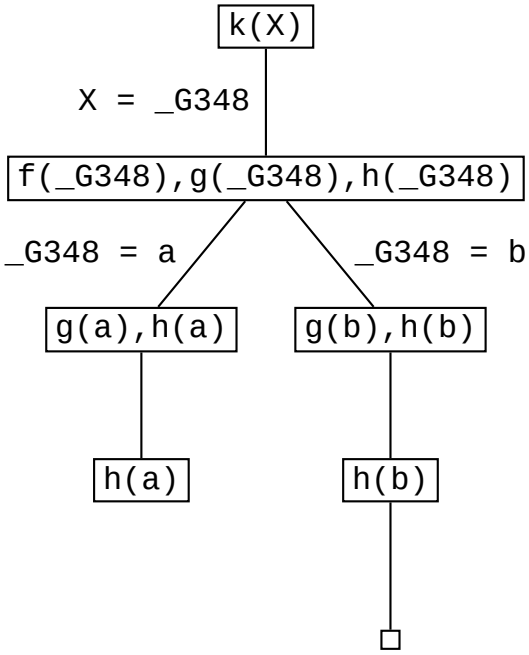
```
g(b),h(b).
```

But `g(b)` is a fact in the knowledge base, so this is satisfied too, leaving the goal list:

```
h(b).
```

And this fact too is in the knowledge base, so this goal is also satisfied. Important: *Prolog now has an empty list of goals.* This means that it has proved everything it had to to establish the original goal, namely `k(X)`. So this query is satisfiable, and moreover, Prolog has also discovered what it has to do to satisfy it, namely instantiate `X` to `b`.

Representing these last steps graphically gives us



It is interesting to consider what happens if we then ask for another solution by typing:

```
;
```

This forces Prolog to backtrack to the last choice point, to try and find another possibility. However, there is no other choice point, as there are no other possibilities for matching `h(b)`, `g(b)`, `f(_G348)`, or `k(X)` with clauses in the knowledge base. So at this point Prolog would correctly have said 'no'. Of course, if there had been other rules involving `k`, Prolog would have gone off and tried to use them in exactly the way we have described: that is, by searching top to bottom in the knowledge base, left to right in goal lists, and backtracking to the previous choice point whenever it fails.

Now, look at the graphical representation that we built while searching for proofs of `k(X)`. It is a tree structure. The nodes of the tree say which are the goals that have to be satisfied at a certain point during the search and at the edges we keep track of the variable instantiations that are made when the current goal (i.e. the first one in the list of goals) is matched to a fact or the head of a rule in the knowledge base. Such trees are called search trees and they are a nice way of visualizing the steps that are taken in searching for a proof of some query. Leaf nodes which still contain unsatisfied goals are point where Prolog failed, because it made a wrong decision somewhere along the path. Leaf nodes with an empty goal list correspond to a possible solution. The information on the edges along the path from the root node to that leaf tell you what are the variable instantiations with which the query is satisfied.

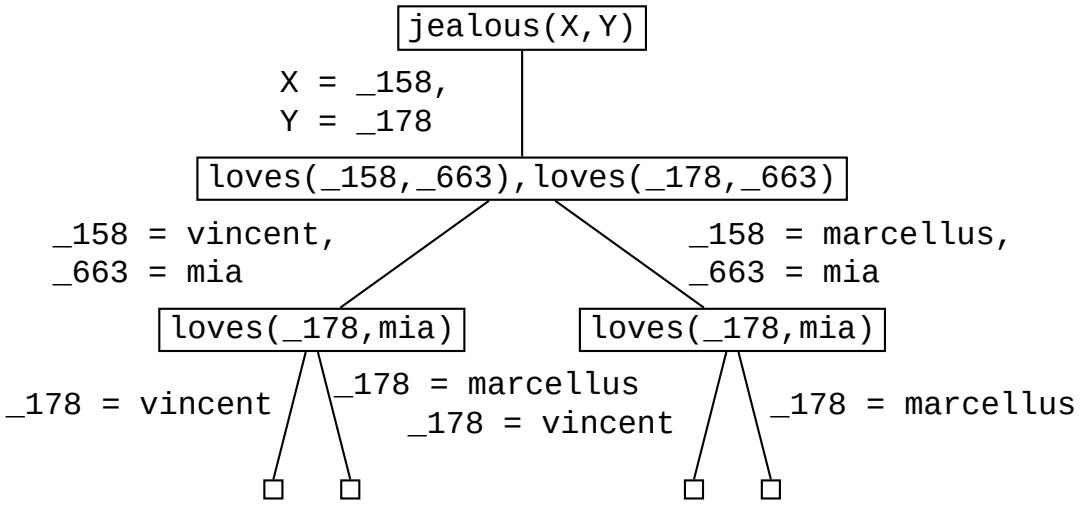
Let's have a look at another example. Suppose that we are working with the following knowledge base:

```
loves(vincent,mia).
loves(marcellus,mia).

jealous(X,Y) :- loves(X,Z),loves(Y,Z).
```

Now, we pose the query

```
?- jealous(X,Y).
```

The search tree for this query looks like this:



There is only one possibility of matching `jealous(X,Y)` against the knowledge base. That is by using the rule

```
jealous(X,Y) :- loves(X,Z),loves(Y,Z).
```

The new goals that have to be satisfied are then

```
loves(_G100,_G101),loves(_G102,_G101).
```

Now, we have to match `loves(_G100,_G101)` against the knowledge base. There are two ways of how this can be done: it can either be matched with the first fact or with the second fact. This is why the path branches at this point. In both cases the goal `loves(_G102,mia)` is left, which also has two possibilities of how it can be satisfied, namely the same ones as above. So, we have four leaf nodes with an empty goal list, which means that there are four ways for satisfying the query. The variable instantiation for each of them can be read off the path from the root to the leaf node. They are

1. `X = \_158 = vincent` and `Y = \_178 = vincent`
2. `X = \_158 = vincent` and `Y = \_178 = marcellus`
3. `X = \_158 = marcellus` and `Y = \_178 = vincent`
4. `X = \_158 = marcellus` and `Y = \_178 = marcellus`

3

Recursion

This lecture has two main goals:

1. To introduce **recursive definitions** in Prolog.
2. To show that there can be mismatches between the **declarative** meaning of a Prolog program, and its **procedural** meaning.

3.1 Recursive definitions

Predicates can be defined recursively. Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself.

3.1.1 Example 1: Eating

Consider the following knowledge base:

```
is_digesting(X,Y) :- just_ate(X,Y).
is_digesting(X,Y) :-
        just_ate(X,Z),
        is_digesting(Z,Y).

just_ate(mosquito,blood(john)).
just_ate(frog,mosquito).
just_ate(stork,frog).
```

At first glance this seems pretty ordinary: it's just a knowledge base containing two facts and two rules. But the definition of the `is_digesting/2` predicate is *recursive*. Note that `is_digesting` is (at least partially) defined in terms of itself, for the `is_digesting` functor occurs on both the left and right hand sides of the second rule. Crucially, however, there is an 'escape' from this circularity. This is provided by the `just_ate` predicate, which occurs in both the first and second rules. (Significantly, the right hand side of the first rule makes no mention of `is_digesting`.) Let's now consider both the **declarative** and **procedural** meanings of this rule.

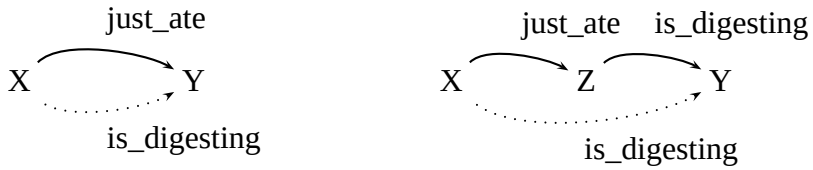
The word *declarative* is used to talk about the logical meaning of Prolog knowledge bases. That is, the declarative meaning of a Prolog knowledge base is simply 'what it says', or 'what it means, if we read it as a collection of logical statements'. And the declarative meaning of this recursive definition is fairly straightforward. The first clause (the 'escape' clause, the one that is not recursive, or as we shall usually call it, the **base** clause), simply says that: *if* `X` has just eaten `Y`, *then* `X` is now digesting `Y`. This is obviously a sensible definition.

So what about the second clause, the **recursive** clause? This says that: *if* `X` has just eaten `Z` *and* `Z` is digesting `Y`, *then* `X` is digesting `Y`, too. Again, this is obviously a sensible definition.

So now we know what this recursive definition *says*, but what happens when we pose a query that actually needs to *use* this definition? That is, what does this definition actually *do*? To use the normal Prolog terminology, what is its *procedural* meaning?

This is also reasonably straightforward. The base rule is like all the earlier rules we've seen. That is, if we ask whether `X` is digesting `Y`, Prolog can use this rule to ask instead the question: has `X` just eaten `Y`?

What about the recursive clause? This gives Prolog another strategy for determining whether `X` is digesting `Y`: *it can try to find some* `Z` *such that* `X` *has just eaten* `Z`, and `Z` *is digesting* `Y`. That is, this rule lets Prolog break the task apart into two subtasks. Hopefully, doing so will eventually lead to simple problems which can be solved by simply looking up the answers in the knowledge base. The following picture sums up the situation:



Let's see how this works. If we pose the query:

```
?- is_digesting(stork,mosquito).
```

then Prolog goes to work as follows. First, it tries to make use of the first rule listed concerning `is_digesting`; that is, the base rule. This tells it that `X` is digesting `Y` if `X` just ate `Y`, By unifying `X` with `stork` and `Y` with `mosquito` it obtains the following goal:

```
just_ate(stork,mosquito).
```

But the knowledge base doesn't contain the information that the stork just ate the mosquito, so this attempt fails. So Prolog next tries to make use of the second rule. By unifying `X` with `stork` and `Y` with `mosquito` it obtains the following goals:

```
just_ate(stork,Z),  is_digesting(Z,mosquito).
```

That is, to show `is_digesting(stork,mosquito)`, Prolog needs to find a value for `Z` such that, firstly, `just_ate(stork,Z)` and secondly, `is_digesting(Z,mosquito)` And there *is* such a value for `Z`, namely `frog`. It is immediate that

```
just_ate(stork,frog)
```

will succeed, for this fact is listed in the knowledge base. And deducing

```
is_digesting(frog,mosquito)
```

is almost as simple, for the first clause of `is_digesting/2` reduces this goal to deducing

```
just_ate(frog,mosquito)
```

and this is a fact listed in the knowledge base.

Well, that's our first example of a recursive rule definition. We're going to learn a lot more about them in the next few weeks, but one very practical remark should be made right away. Hopefully it's clear that when you write a recursive predicate, it should always have at least two clauses: a base clause (the clause that stops the recursion at some point), and one that contains the recursion. If you don't do this, Prolog can spiral off into an unending sequence of useless computations. For example, here's an extremely simple example of a recursive rule definition:

```
p :- p.
```

That's it. Nothing else. It's beautiful in its simplicity. And from a declarative perspective it's an extremely sensible (if rather boring) definition: it says 'if property p holds, then property p holds'. You can't argue with that.

But from a procedural perspective, this is a wildly dangerous rule. In fact, we have here the ultimate in dangerous recursive rules: exactly the same thing on both sides, and no base clause to let us escape. For consider what happens when we pose the following query:

```
?- p.
```

Prolog asks itself: how do I prove `p`? And it realizes, 'Hey, I've got a rule for that! To prove `p` I just need to prove `p`!'. So it asks itself (again): how do I prove `p`? And it realizes, 'Hey, I've got a rule for that! To prove `p` I just need to prove `p`!'. So it asks itself (yet again): how do I prove `p`? And it realizes, 'Hey, I've got a rule for that! To prove `p` I just need to prove `p`!" So then it asks itself (for the fourth time): how do I prove `p`? And it realizes that...

If you make this query, Prolog won't answer you: it will head off, looping desperately away in an unending search. That is, it won't terminate, and you'll have to interrupt it. Of course, if you use `trace`, you can step through one step at a time, until you get sick of watching Prolog loop.

3.1.2 Example 2: Descendant

Now that we know something about *what* recursion in Prolog involves, it is time to ask *why* it is so important. Actually, this is a question that can be answered on a number of levels, but for now, let's keep things fairly practical. So: when it comes to writing useful Prolog programs, are recursive definitions really so important? And if so, why?

Let's consider an example. Suppose we have a knowledge base recording facts about the child relation:

```
child(charlotte,caroline).
child(caroline,laura).
```

That is, Caroline is a child of Charlotte, and Laura is a child of Caroline. Now suppose we wished to define the descendant relation; that is, the relation of being a child of, or a child of a child of, or a child of a child of a child of, or.... Here's a first attempt to do this. We could add the following two *non*-recursive rules to the knowledge base:

```
descend(X,Y) :- child(X,Y).

descend(X,Y) :- child(X,Z),
                child(Z,Y).
```

Now, fairly obviously these definitions work up to a point, but they are clearly extremely limited: they only define the concept of descendant-of for two generations or less. That's ok for the above knowledge base, but suppose we get some more information about the child-of relation and we expand our list of child-of facts to this:

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).
```

Now our two rules are inadequate. For example, if we pose the queries

```
?- descend(martha,laura).
```

or

```
?- descend(charlotte,rose).
```

we get the answer 'No!', which is *not* what we want. Sure, we could 'fix' this by adding the following two rules:

```
descend(X,Y) :- child(X,Z_1),
                child(Z_1,Z_2),
                child(Z_2,Y).

descend(X,Y) :- child(X,Z_1),
                child(Z_1,Z_2),
                child(Z_2,Z_3),
                child(Z_3,Y).
```

But, let's face it, this is clumsy and hard to read. Moreover, if we add further child-of facts, we could easily find ourselves having to add more and more rules as our list of child-of facts grow, rules like:

```
descend(X,Y) :- child(X,Z_1),
                child(Z_1,Z_2),
                child(Z_2,Z_3),
                    ⋮
                child(Z_17,Z_18).
                child(Z_18,Z_19).
                child(Z_19,Y).
```

This is not a particularly pleasant (or sensible) way to go!

But we don't need to do this at all. We can avoid having to use ever longer rules entirely. The following recursive rule fixes everything exactly the way we want:

```
descend(X,Y) :- child(X,Y).

descend(X,Y) :- child(X,Z),
                descend(Z,Y).
```

What does this say? The declarative meaning of the base clause is: *if* `Y` is a child of `X`, *then* `Y` is a descendant of `X`. Obviously sensible.

So what about the recursive clause? Its declarative meaning is: *if* `Z` is a child of `X`, *and* `Y` is a descendant of `Z`, *then* `Y` is a descendant of `X`. Again, this is obviously true.

So let's now look at the procedural meaning of this recursive predicate, by stepping through an example. What happens when we pose the query:

```
?- descend(martha,laura)
```

Prolog first tries the first rule. The variable `X` in the head of the rule is unified with `martha` and `Y` with laura and the next goal Prolog tries to prove is

```
child(martha,laura)
```

This attempt fails, however, since the knowledge base neither contains the fact `child(martha,laura)` nor any rules that would allow to infer it. So Prolog backtracks and looks for an alternative way of proving `descend(martha,laura)`. It finds the second rule in the knowledge base and now has the following subgoals:

```
child(martha,_633),descend(_633,laura).
```

Prolog takes the first subgoal and tries to match it onto something in the knowledge base. It finds the fact `child(martha,charlotte)` and the variable `_633` gets instantiated to `charlotte`. Now that the first subgoal is satisfied, Prolog moves to the second subgoal. It has to prove

```
descend(charlotte,laura).
```

This is the recursive call of the predicate `descend/2`. As before, Prolog starts with the first rule, but fails, because the goal

```
child(charlotte,laura)
```

cannot be proved. Backtracking, Prolog finds that there is a second possibility to be checked for `descend(charlotte,laura)`, viz. the second rule, which again gives Prolog two new subgoals:

```
child(charlotte,_1785),descend(_1785,laura).
```

The first subgoal can be unified with the fact `child(charlotte,caroline)` of the knowledge base, so that the variable `_1785` is instantiated with `caroline`. Next Prolog tries to prove

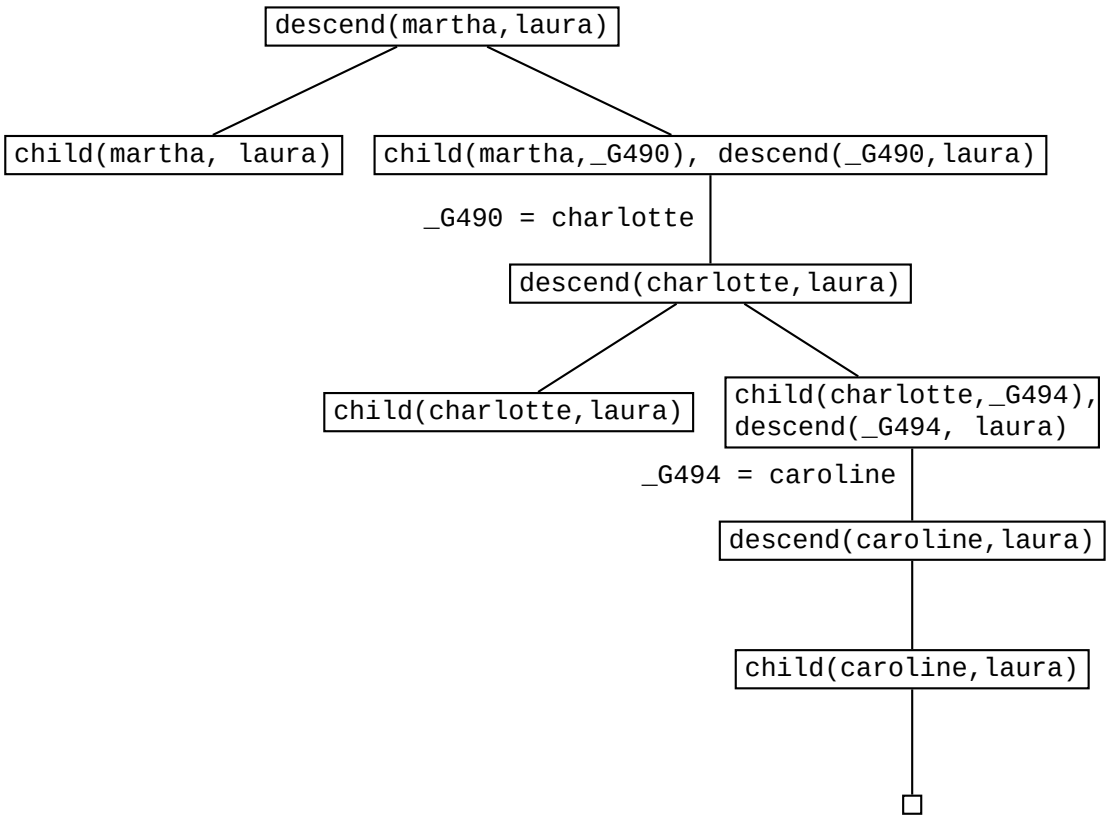
```
descend(caroline,laura).
```

This is the second recursive call of predicate `descend/2`. As before, it tries the first rule first, obtaining the following new goal:

```
child(caroline,laura)
```

This time Prolog succeeds, since `child(caroline,laura)` is a fact in the database. Prolog has found a proof for the goal `descend(caroline,laura)` (the second recursive call). But this means that `child(charlotte,laura)` (the first recursive call) is also true, which means that our original query `descend(martha,laura)` is true as well.

Here is the search tree for the query `descend(martha,laura)`. Make sure that you understand how it relates to the discussion in the text; i.e. how Prolog traverses this search tree when trying to prove this query.



It should be obvious from this example that no matter how many generations of children we add, we will always be able to work out the descendant relation. That is, the recursive definition is both general and compact: it contains *all* the information in the previous rules, and much more besides. In particular, the previous lists of non-recursive rules only defined the descendant concept up to some fixed number of generations: we would need to write down *infinitely many* non-recursive rules if we wanted to capture this concept fully, and of course that's impossible. But, in effect, that's what the recursive rule does for us: it bundles up all this information into just three lines of code.

Recursive rules are really important. They enable to pack an enormous amount of information into a compact form and to define predicates in a natural way. Most of the work you will do as a Prolog programmer will involve writing recursive rules.

3.1.3 Example 3: Successor

In the previous lectures we remarked that *building structure through matching* is a key idea in Prolog programming. Now that we know about recursion, we can give more interesting illustrations of this.

Nowadays, when human beings write numerals, they usually use *decimal* notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on) but as you probably know, there are many other notations. For example, because computer hardware is generally based on digital circuits, computers usually use *binary* notation to represent numerals (0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on), for the 0 can be implemented as a switch being off, the 1 as a switch being on. Other cultures use different systems. For example, the ancient Babylonians used a base 60 system, while the ancient Romans used a rather ad-hoc system (I, II, III, IV, V, VI, VII, VIII, IX, X). This last example shows that notational issues can be important. If you don't believe this, try figuring out a systematic way of doing long-division in Roman notation. As you'll discover, it's a frustrating task. In fact, the Romans had a group of professionals (analogs of modern accountants) who specialized in this.

Well, here's yet another way of writing numerals, which is sometimes used in mathematical logic. It makes use of just four symbols: 0, *succ*, and the left and right parentheses. This style of numeral is defined by the following inductive definition:

1. 0 is a numeral.
2. If *X* is a numeral, then so is *succ(X)*.

As is probably clear, *succ* can be read as short for *successor*. That is, *succ(X)* represents the number obtained by adding one to the number represented by *X*. So this is a very simple notation: it simply says that 0 is a numeral, and that all other numerals are built by stacking *succ* symbols in front. (In fact, it's used in mathematical logic because of this simplicity.

Although it wouldn't be pleasant to do household accounts in this notation, it is a very easy notation to prove things *about*.) Now, by this stage it should be clear that we can turn this definition into a Prolog program. The following knowledge base does this:

```
numeral(0).

numeral(succ(X)) :- numeral(X).
```

So if we pose queries like

```
?- numeral(succ(succ(succ(0)))).
```

we get the answer 'yes'. But we can do some more interesting things. Consider what happens when we pose the following query:

```
?- numeral(X).
```

That is, we're saying 'Ok, show me some numerals'. Then we can have the following dialogue with Prolog:

```
X = 0 ;

X = succ(0) ;

X = succ(succ(0)) ;

X = succ(succ(succ(0))) ;

X = succ(succ(succ(succ(0)))) ;

X = succ(succ(succ(succ(succ(0))))) ;

X = succ(succ(succ(succ(succ(succ(0)))))) ;

X = succ(succ(succ(succ(succ(succ(succ(0))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(0)))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))))))))))))))) ;

X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))))))))))))))))
yes
```

Yes, Prolog is counting: but what's really important is *how* it's doing this. Quite simply, it's backtracking through the recursive definition, and actually *building* numerals using matching. This is an instructive example, and it is important that you understand it. The best way to do so is to sit down and try it out, with `trace` turned on.

Building and binding. Recursion, matching, and proof search. These are ideas that lie at the heart of Prolog programming. Whenever we have to generate or analyze recursively structured objects (such as these numerals) the interplay of these ideas makes Prolog a powerful tool. For example, in the next lecture we introduce **lists**, an extremely important recursive data structure, and we will see that Prolog is a natural list processing language. Many applications (computational linguistics is a prime example) make heavy use of recursively structured objects, such as trees and feature structures. So it's not particularly surprising that Prolog has proved useful in such applications.

3.1.4 Example 3: Addition

As a final example, let's see whether we can use the representation of numerals that we introduced in the previous section for doing simple arithmetic. Let's try to define addition. That is, we want to define a predicate `add/3` which when given two numerals as the first and second argument returns the result of adding them up as its third argument. E.g.,

```
?- add(succ(succ(0)),succ(succ(0)),succ(succ(succ(succ(0))))).
yes
?- add(succ(succ(0)),succ(0),Y).
Y = succ(succ(succ(0)))
```

There are two things which are important to notice. First, whenever the first argument is `0`, the third argument has to be the same as the second argument:

```
?- add(0,succ(succ(0)),Y).
Y = succ(succ(0))
?- add(0,0,Y).
Y = 0
```

This is the case that we want to use for the base clause.

Secondly, assume that we want to add the two numerals `X` and `Y` (e.g. `succ(succ(succ(0)))` and `succ(succ(0))`) and that `X` is not `0`. Now, if `X'` is the numeral that has one `succ` functor less than `X` (i.e. `succ(succ(0))` in our example) and if we know the result – let's call it `Z` – of adding `X'` and `Y` (namely `succ(succ(succ(succ(0))))`), then it is very easy to compute the result of adding `X` and `Y`: we just have to add one `succ`-functor to `Z`. This is what we want to express with the recursive clause.

Here is the predicate definition that expresses exactly what we just said:

```
add(0,Y,Y).

add(succ(X),Y,succ(Z)) :-
        add(X,Y,Z).
```

So, what happens, if we give Prolog this predicate definition and then ask:

```
?- add(succ(succ(succ(0))), succ(succ(0)), R)
```

Let's go through the way Prolog processes this query step by step. The trace and the search tree are given below.

The first argument is not `0` which means that only the second clause for `add` matches. This leads to a recursive call of `add`. The outermost `succ` functor is stripped off the first argument of the original query, and the result becomes the first argument of the recursive query. The second argument is just passed on to the recursive query, and the third argument of the recursive query is a variable, the internal variable `_G648` in the trace given below. `_G648` is not instantiated, yet. However, it is related to `R` (which is the variable that we had as third argument in the original query), because `R` was instantiated to `succ(_G648)`, when the query was matched to the head of the second clause. But that means that `R` is not a completely uninstantiated variable anymore. It is now a complex term, that has a (uninstantiated) variable as its argument.

The next two steps are essentially the same. With every step the first argument becomes one level smaller. The trace and the search tree show this nicely. At the same time one `succ` functor is added to `R` with every step, but always leaving the argument of the innermost variable uninstantiated. After the first recursive call `R` is `succ(_G648)`, in the second recursive call `_G648` is instantiated with `succ(_G650)`, so that `R` is `succ(succ(_G650)`, in the third recursive call `_G650` is instantiated with `succ(_G652)` and `R` therefore becomes `succ(succ(succ(_G652)))`. The search tree shows this step by step instantiation.

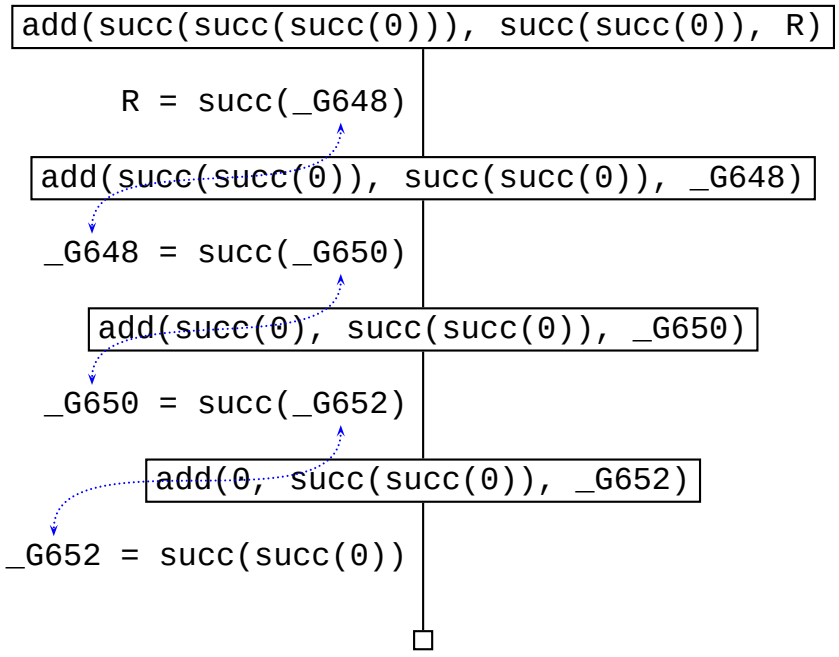
At some point all `succ` functors have been stripped off the first argument and we have reached the base clause. Here, the third argument is equated with the second argument, so that "the hole" in the complex term `R` is finally filled.

This is a trace for the query `add(succ(succ(succ(0))), succ(succ(0)), R)`:

```
Call: (6) add(succ(succ(succ(0))), succ(succ(0)), R)

Call: (7) add(succ(succ(0)), succ(succ(0)), _G648)

Call: (8) add(succ(0), succ(succ(0)), _G650)
```

```
Call: (9) add(0, succ(succ(0)), _G652)

Exit: (9) add(0, succ(succ(0)), succ(succ(0)))

Exit: (8) add(succ(0), succ(succ(0)), succ(succ(succ(0))))

Exit: (7) add(succ(succ(0)), succ(succ(0)),
                                        succ(succ(succ(succ(0)))))

Exit: (6) add(succ(succ(succ(0))), succ(succ(0)),
                                    succ(succ(succ(succ(succ(0))))))
```

And here is the search tree for this query:



3.2 Clause ordering, goal ordering, and termination

Prolog was the first reasonably successful attempt to make a *logic programming* language. Underlying logic programming is a simple (and seductive) vision: the task of the programmer is simply to *describe* problems. The programmer should write down (in the language of logic) a declarative specification (that is: a knowledge base), which describes the situation of interest. The programmer shouldn't have to tell the computer *what* to do. To get

information, he or she simply asks the questions. It's up to the logic programming system to figure out how to get the answer.

Well, that's the idea, and it should be clear that Prolog has taken some interesting steps in this direction. But Prolog is *not*, repeat *not*, a full logic programming language. If you only think about the declarative meaning of a Prolog program, you are in for a very tough time. As we learned in the previous lecture, Prolog has a very specific way of working out the answer to queries: it searches the knowledge base from top to bottom, clauses from left to right, and uses backtracking to recover from bad choices. These procedural aspects have an important influence on what actually happens when you make a query. We have already seen a dramatic example of a mismatch between procedural and declarative meaning of a knowledge base (remember the `p:- p` program?), and as we shall now see, it is easy to define knowledge bases with the same declarative meaning, but very different procedural meanings.

Recall our earlier descendant program (let's call it `descend1.pl`):

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y) :- child(X,Y).

descend(X,Y) :- child(X,Z),
                descend(Z,Y).
```

We'll make two changes to it, and call the result `descend2.pl`:

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y) :- descend(Z,Y),
                child(X,Z).

descend(X,Y) :- child(X,Y).
```

From a declarative perspective, what we have done is very simple: we have merely reversed the order of the two rules, and reversed the order of the two goals in the recursive clause.

So, viewed as a purely logical definition, nothing has changed. We have *not* changed the declarative meaning of the program.

But the procedural meaning has changed dramatically. For example, if you pose the query

```
?- descend(martha,rose).
```

you will get an error message ('out of local stack', or something similar). Prolog is looping. Why? Well, to satisfy `descend(martha,rose).` Prolog uses the first rule. This means that its next goal will be to satisfy the query

```
descend(W1,rose)
```

for some new variable `W1`. But to satisfy this new goal, Prolog again has to use the first rule, and this means that its next goal is going to be

```
descend(W2,rose)
```

for some new variable `W2`. And of course, this in turn means that its next goal is going to be `descend(W3,rose)` and then `descend(W4,rose)`, and so on.

In short, `descend1.pl` and `descend2.pl` are Prolog knowledge bases with the same declarative meaning but different procedural meanings: from a purely logical perspective they are identical, but they behave very differently.

Let's look at another example. Recall our earlier successor program (let's call it `numeral1.pl`):

```
numeral(0).

numeral(succ(X)) :- numeral(X).
```

Now, we simply swap the order of the two clauses and call the result `numeral2.pl`:

```
numeral(succ(X)) :- numeral(X).

numeral(0).
```

Clearly the declarative, or logical, content of this program is exactly the same as the earlier version. But what about its behavior?

Ok, if we pose a query about *specific* numerals, `numeral2.pl` will terminate with the answer we expect. For example, if we ask:

```
?- numeral(succ(succ(succ(0)))).
```

we will get the answer ‘yes’. But if we try to *generate* numerals, that is, if we give it the query

```
?- numeral(X).
```

the program won’t halt. Make sure you understand why not. Once again, we have two knowledge bases with the same declarative meaning but different procedural meanings.

Because the declarative and procedural meanings of a Prolog program can differ, when writing Prolog programs you need to bear both aspects in mind. Often you can get the overall idea (‘the big picture’) of how to write the program by thinking declaratively, that is, by thinking simply in terms of describing the problem accurately. But then you need to think about how Prolog will actually evaluate queries. Are the rule orderings sensible? How will the program actually run? Learning to flip back and forth between procedural and declarative questions is an important part of learning to program in Prolog.

4

Lists

This lecture has two main goals:

1. To introduce **lists**, an important recursive data structure widely used in computational linguistics.
2. To define **member**, a fundamental Prolog tool for manipulating lists, and to introduce the idea of **recursing down lists**.

4.1 Lists

As its name suggests, a list is just a plain old list of items. Slightly more precisely, it is a finite sequence of elements. Here are some examples of lists in Prolog:

```
[mia, vincent, jules, yolanda]

[mia, robber(honey_bunny), X, 2, mia]

[]

[mia, [vincent, jules], [butch, girlfriend(butch)]]

[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]]
```

We can learn some important things from these examples.

First, we can specify lists in Prolog by enclosing the **elements** of the list in square brackets (that is, the symbols `[` and `]`). The elements are separated by commas. For example, the first list `[mia, vincent, jules, yolanda]` has four elements, namely `mia`, `vincent`,

`jules`, and `yolanda`. The **length** of a list is the number of elements it has, so our first example is a list of length four.

From our second example, `[mia,robber(honey_bunny),X,2,mia]`, we learn that all sorts of Prolog objects can be elements of a list. The first element of this list is `mia`, an atom; the second element is `robber(honey_bunny)`, a complex term; the third element is `X`, a variable; the fourth element is `2`, a number. Moreover, we also learn that the same item may occur more than once in the same list: for example, the fifth element of this list is `mia`, which is same as the first element.

The third example shows that there is a very special list, the **empty list**. The empty list (as its name suggests) is the list that contains no elements. What is the length of the empty list? Zero, of course (for the length of a list is the number of members it contains, and the empty list contains nothing).

The fourth example teaches us something extremely important: lists can contain other lists as elements. For example, the second element of

```
[mia, [vincent, jules], [butch,girlfriend(butch)]
```

is the list `[vincent,jules]`, and the third element is `[butch,girlfriend(butch)]]`. In short, lists are examples of **recursive data structures**: lists can be made out of lists. What is the length of the fourth list? The answer is: three. If you thought it was five (or indeed, anything else) you're not thinking about lists in the right way. The elements of the list are the things between the outermost square brackets separated by commas. So this list contains *three* elements: the first element is `mia`, the second element is `[vincent, jules]`, and the third element is `[butch, girlfriend(butch)]`.

The last example mixes all these ideas together. We have here a list which contains the empty list (in fact, it contains it twice), the complex term `dead(zed)`, two copies of the list `[2, [b, chopper]]`, and the variable `Z`. Note that the third (and the last) elements are lists which themselves contain lists (namely `[b, chopper]`).

Now for a very important point. Any non-empty list can be thought of as consisting of two parts: the **head** and the **tail**. The head is simply the first item in the list; the tail is everything else. Or more precisely, the tail is the list that remains when we take the first element away, i.e. *the tail of a list is always a list* again. For example, the head of

```
[mia, vincent, jules, yolanda]
```

is `mia` and the tail is `[vincent, jules, yolanda]`. Similarly, the head of

```
[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]]
```

is `[]`, and the tail is `[dead(zed), [2,[b,chopper]],[],Z,[2,[b, chopper]]]`. And what are the head and the tail of the list `[dead(zed)]`? Well, the head is the first element of the list, which is `dead(zed)`, and the tail is the list that remains if we take the head away, which, in this case, is the empty list `[]`.

Note that only non-empty lists have heads and tails. That is, the empty list contains no internal structure. For Prolog, the empty list `[]` is a special, particularly simple, list.

Prolog has a special built-in operator `|` which can be used to decompose a list into its head and tail. It is *very important* to get to know how to use `|`, for it is a key tool for writing Prolog list manipulation programs.

The most obvious use of `|` is to extract information from lists. We do this by using `|` together with matching. For example, to get hold of the head and tail of `[mia,vincent, jules,yolanda]` we can pose the following query:

```
?- [Head| Tail] = [mia, vincent, jules, yolanda].

Head = mia
Tail = [vincent,jules,yolanda]
yes
```

That is, the head of the list has become bound to `Head` and the tail of the list has become bound to `Tail`. Note that there is nothing special about `Head` and `Tail`, they are simply variables. We could just as well have posed the query:

```
?- [X|Y] = [mia, vincent, jules, yolanda].

X = mia
Y = [vincent,jules,yolanda]
yes
```

As we mentioned above, only non-empty lists have heads and tails. If we try to use `|` to pull `[]` apart, Prolog will fail:

```
?- [X|Y] = [].

no
```

That is, Prolog treats `[]` as a special list. This observation is *very* important. We'll see why later.

Let's look at some other examples. We can extract the head and tail of the following list just as we saw above:

```
?- [X|Y] = [[], dead(zed), [2, [b, chopper]], [], Z].

X = []
Y = [dead(zed),[2,[b,chopper]],[],_7800]
Z = _7800
yes
```

That is: the head of the list is bound to `X`, the tail is bound to `Y`. (We also get the information that Prolog has bound `Z` to the internal variable `_7800`.)

But we can can do a lot more with `|`; it really is a very flexible tool. For example, suppose we wanted to know what the first *two* elements of the list were, and also the remainder of the list after the second element. Then we'd pose the following query:

```
?- [X,Y | W] = [[], dead(zed), [2, [b, chopper]], [], Z].

X = []
Y = dead(zed)
W = [[2,[b,chopper]],[],_8327]
Z = _8327
yes
```

That is: the head of the list is bound to `X`, the second element is bound to `Y`, and the remainder of the list after the second element is bound to `W`. `W` is the list that remains when we take away the first two elements. So, `|` can not only be used to split a list into its head and its tail, but we can in fact use it to split a list at any point. Left of the `|`, we just have to enumerate how many elements we want to take away from the beginning of the list, and right of the `|` we will then get what remains of the list. In this example, we also get the information that Prolog has bound `Z` to the internal variable `_8327`.

This is a good time to introduce the **anonymous variable**. Suppose we were interested in getting hold of the second and fourth elements of the list:

```
[[], dead(zed), [2, [b, chopper]], [], Z].
```

Now, we *could* find out like this:

```
?- [X1,X2,X3,X4 | Tail] = [[], dead(zed), [2, [b, chopper]], [], Z].

X1 = []
X2 = dead(zed)
X3 = [2,[b,chopper]]
```

```
X4 = []
Tail = [_8910]
Z = _8910
yes
```

OK, we have got the information we wanted: the values we are interested in are bound to the variables `X2` and `X4`. But we've got a lot of other information too (namely the values bound to `X1`, `X3` and `Tail`). And perhaps we're not interested in all this other stuff. If so, it's a bit silly having to explicitly introduce variables `X1`, `X3` and `Tail` to deal with it. And in fact, there is a simpler way to obtain *only* the information we want: we can pose the following query instead:

```
?- [_,X,_,Y|_] = [[], dead(zed), [2, [b, chopper]], [], Z].

X = dead(zed)
Y = []
Z = _9593
yes
```

The _ symbol (that is, *underscore*) is the anonymous variable. We use it when we need to use a variable, but we're not interested in what Prolog instantiates it to. As you can see in the above example, Prolog didn't bother telling us what _ was bound to. Moreover, note that each occurrence of _ is *independent*: each is bound to something different. This couldn't happen with an ordinary variable of course, but then the anonymous variable isn't meant to be ordinary. It's simply a way of telling Prolog to bind something to a given position, completely independently of any other bindings.

Let's look at one last example. The third element of our working example is a list (namely `[2, [b, chopper]]`). Suppose we wanted to extract the tail of this internal list, and that we are not interested in any other information. How could we do this? As follows:

```
?- [_,_,[_|X]|_] =
      [[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]].

X = [[b,chopper]]
Z = _10087
yes
```

4.2 Member

It's time to look at our first example of a Prolog program for manipulating lists. One of the most basic things we would like to know is whether something is an element of a list or not.

So let's write a program that, when given as inputs an arbitrary object *X* and a list *L*, tells us whether or not *X* belongs to *L*. The program that does this is usually called **member**, and it is the simplest example of a Prolog program that exploits the recursive structure of lists. Here it is:

```
member(X,[X|T]).

member(X,[H|T]) :- member(X,T).
```

That's all there is to it: one fact (the first clause) and one rule (the second clause). But note that the rule is recursive (after all, the functor `member` occurs in both the rule's head and tail) and it is this that explains why such a short program is all that is required. Let's take a closer look.

We'll start by reading the program declaratively. And read this way, it is obviously sensible. The first clause (the fact) simply says: an object `X` is a member of a list if it is the head of that list. Note that we used the built-in `|` operator to state this (simple but important) principle about lists.

What about the second clause, the recursive rule? This says: an object `X` is member of a list if it is a member of the tail of the list. Again, note that we used the `|` operator to state this principle.

Now, clearly this definition makes good declarative sense. But does this program actually *do* what it is supposed to do? That is, will it really tell us whether an object `X` belongs to a list `L`? And if so, how exactly does it do this? To answer such questions, we need to think about its procedural meaning. Let's work our way through a few examples.

Suppose we posed the following query:

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
```

Prolog will immediately answer 'Yes'. Why? Because it can unify `yolanda` with both occurrences of `X` in the first clause (the fact) in the definition of `member/2`, so it succeeds immediately.

Now consider the following query:

```
?- member(vincent,[yolanda,trudy,vincent,jules]).
```

Now the first rule won't help (`vincent` and `yolanda` are distinct atoms) so Prolog goes to the second clause, the recursive rule. This gives Prolog a new goal: it now has to see if

```
member(vincent,[trudy,vincent,jules])
```

holds. Once again the first clause won't help, so Prolog goes (again) to the recursive rule. This gives it a new goal, namely

```
member(vincent,[vincent,jules]).
```

This time, the first clause does help, and the query succeeds.

So far so good, but we need to ask an important question. What happens when we pose a query that *fails*? For example, what happens if we pose the query

```
?- member(zed,[yolanda,trudy,vincent,jules]).
```

Now, this should obviously fail (after all, `zed` is not on the list). So how does Prolog handle this? In particular, how can we be sure that Prolog really will *stop*, and say *no*, instead going into an endless recursive loop?

Let's think this through systematically. Once again, the first clause cannot help, so Prolog uses the recursive rule, which gives it a new goal

```
member(zed,[trudy,vincent,jules]).
```

Again, the first clause doesn't help, so Prolog reuses the recursive rule and tries to show that

```
member(zed,[vincent,jules]).
```

Similarly, the first rule doesn't help, so Prolog reuses the second rule yet again and tries the goal

```
member(zed,[jules]).
```

Again the first clause doesn't help, so Prolog uses the second rule, which gives it the goal

```
member(zed,[])
```

And *this* is where things get interesting. Obviously the first clause can't help here. But note: *the recursive rule can't do anything more either*. Why not? Simple: the recursive rule relies on splitting the list into a head and a tail, but as we have already seen, the empty list *can't* be split up in this way. So the recursive rule cannot be applied either, and Prolog stops searching for more solutions and announces 'No'. That is, it tells us that `zed` does not belong to the list, which is, of course, what it ought to do.

We could summarize the `member/2` predicate as follows. It is a recursive predicate, which systematically searches down the length of the list for the required item. It does this by stepwise breaking down the list into smaller lists, and looking at the first item of each smaller list. This mechanism that drives this search is recursion, and the reason that this recursion is safe (that is, the reason it does not go on forever) is that at the end of the line Prolog has to ask a question about the empty list. The empty list *cannot* be broken down into smaller parts, and this allows a way out of the recursion.

Well, we've now seen why `member/2` works, but in fact it's far more useful than the previous example might suggest. Up till now we've only been using it to answer yes/no questions. But we can also pose questions containing variables. For example, we can have the following dialog with Prolog:

```
?- member(X,[yolanda,trudy,vincent,jules]).

X = yolanda ;

X = trudy ;

X = vincent ;

X = jules ;

no
```

That is, Prolog has told us what every member of a list is. This is a very common use of `member/2`. In effect, by using the variable we are saying to Prolog: 'Quick! Give me some element of the list!'. In many applications we need to be able to extract members of a list, and this is the way it is typically done.

One final remark. The way we defined `member/2` above is certainly correct, but in one respect it is a little messy.

Think about it. The first clause is there to deal with the head of the list. But although the tail is irrelevant to the first clause, we named the tail using the variable `T`. Similarly, the recursive rule is there to deal with the tail of the list. But although the head is irrelevant here, we named it using the variable `H`. These unnecessary variable names are distracting: it's better to write predicates in a way that focuses attention on what is really important in each clause, and the anonymous variable gives us a nice way of doing this. That is, we can rewrite `member/2` as follows:

```
member(X,[X|_]).

member(X,[_|T]) :- member(X,T).
```

This version is exactly the same, both declaratively and procedurally. But it's just that little bit clearer: when you read it, you are forced to concentrate on what is essential.

4.3 Recursing down lists

Member works by recursively working down a list, doing something to the head, and then recursively doing the same thing to the tail. Recursing down a list (or indeed, several lists) in this way is extremely common in Prolog: so common, in fact, that it is important that you really master the idea. So let's look at another example of the technique at work.

When working with lists, we often want to compare one list with another, or to copy bits of one list into another, or to translate the contents of one list into another, or something similar. Here's an example. Let's suppose we need a predicate `a2b/2` that takes two lists as arguments, and succeeds if the first argument is a list of `a`s, and the second argument is a list of `b`s of exactly the same length. For example, if we pose the following query

```
?- a2b([a,a,a,a],[b,b,b,b]).
```

we want Prolog to say 'yes'. On the other hand, if we pose the query

```
?- a2b([a,a,a,a],[b,b,b]).
```

or the query

```
?- a2b([a,c,a,a],[b,b,5,4]).
```

we want Prolog to say 'no'.

When faced with such tasks, often the best way to set about solving them is to start by thinking about the simplest possible case. Now, when working with lists, 'thinking about the simplest case' often means 'thinking about the empty list', and it certainly means this here. After all: what is the shortest possible list of `a`s? Why, the empty list: it contains no `a`s at all! And what is the shortest possible list of `b`s? Again, the empty list: no `b`s whatsoever in that! So the most basic information our definition needs to contain is

```
a2b([],[]).
```

This records the obvious fact that the empty list contains exactly as many `a`s as `b`s. But although obvious, this fact turns out to play a very important role in our program, as we shall see.

So far so good: but how do we proceed? Here's the idea: for longer lists, *think recursively*. So: when should `a2b/2` decide that two non-empty lists are a list of `a`s and a list of `b`s of exactly the same length? Simple: when the head of the first list is an `a`, and the head of the second list is a `b`, and `a2b/2` decides that the two tails are lists of `a`s and `b`s of exactly the same length! This immediately gives us the following rule:

```
a2b([a|Ta],[b|Tb]) :- a2b(Ta,Tb).
```

This says: the `a2b/2` predicate should succeed if its first argument is a list with head `a`, its second argument is a list with head `b`, and `a2b/2` succeeds on the two tails.

Now, this definition make good sense declaratively. It is a simple and natural recursive predicate, the base clause dealing with the empty list, the recursive clause dealing with non-empty lists. But how does it work in practice? That is, what is its procedural meaning? For example, if we pose the query

```
?- a2b([a,a,a],[b,b,b]).
```

Prolog will say 'yes', which is what we want, by *why* exactly does this happen?

Let's work the example through. In this query, neither list is empty, so the fact does not help. Thus Prolog goes on to try the recursive rule. Now, the query does match the rule (after all, the head of the first list is `a` and the head of the second in `b`) so Prolog now has a new goal, namely

```
a2b([a,a],[b,b]).
```

Once again, the fact does not help with this, but the recursive rule can be used again, leading to the following goal:

```
a2b([a],[b]).
```

Yet again the fact does not help, but the recursive rule does, so we get the following goal:

```
a2b([],[]).
```

At last we can use the fact: this tells us that, yes, we really do have two lists here that contain exactly the same number of `a`s and `b`s (namely, none at all). And because this goal succeeds, this means that the goal

```
a2b([a],[b])
```

succeeds too. This in turn means that the goal

```
a2b([a,a],[b,b])
```

succeeds, and thus that the original goal

```
a2b([a,a,a],[b,b,b])
```

is satisfied.

We could summarize this process as follows. Prolog started with two lists. It peeled the head off each of them, and checked that they were an `a` and a `b` as required. It then recursively analyzed the tails of both lists. That is, it worked down both tails simultaneously, checking that at each stage the tails were headed by an `a` and a `b`. Why did the process stop? Because at each recursive step we had to work with shorter lists (namely the tails of the lists examined at the previous step) and eventually we ended up with empty lists. At this point, our rather trivial looking fact was able to play a vital role: it said 'yes!'. This halted the recursion, and ensured that the original query succeeded.

It's is also important to think about what happens with queries that *fail*. For example, if we pose the query

```
?- a2b([a,a,a,a],[b,b,b]).
```

Prolog will correctly say 'no'. Why? because after carrying out the 'peel off the head and recursively examine the tail' process three times, it will be left with the query

```
a2b([a],[]).
```

But this goal cannot be satisfied. And if we pose the query

```
?- a2b([a,c,a,a],[b,b,5,4]).
```

after carrying out the 'peel off the head and recursively examine the tail' process once, Prolog will have the goal

```
a2b([c,a,a],[b,5,4]).
```

and again, this cannot be satisfied.

Well, that's how `a2b/2` works in simple cases, but we haven't exhausted its possibilities yet. As always with Prolog, it's a good idea to investigate what happens when variables as used as input. And with `a2b/2` something interesting happens: it acts as a translator, translating lists of `a`s to lists of `b`s, and vice versa. For example the query

```
?- a2b([a,a,a,a],X).
```

yields the response

```
X = [b,b,b,b].
```

That is, the list of `a`s has been translated to a list of `b`s. Similarly, by using a variable in the first argument position, we can use it to translate lists of `b`s to lists of `a`s:

```
?- a2b(X,[b,b,b,b]).
X = [a,a,a,a]
```

And of course, we can use variables in both argument positions:

```
?- a2b(X,Y).
```

Can you work out what happens in this case?

To sum up: `a2b/2` is an extremely simple example of a program that works by recursing its way down a pair of lists. But don't be fooled by its simplicity: the kind of programming it illustrates is fundamental to Prolog. Both its declarative form (a base clause dealing with the empty list, a recursive clause dealing with non-empty lists) and the procedural idea it trades on (do something to the heads, and then recursively do the same thing to the tails) come up again and again in Prolog programming. In fact, in the course of your Prolog career, you'll find that you'll write what is essentially the `a2b/2` predicate, or a more complex variant of it, many times over in many different guises.

5

Arithmetic

This lecture has two main goals:

1. To introduce Prolog's built-in abilities for performing **arithmetic**, and
2. To apply them to simple list processing problems, using **accumulators**.

5.1 Arithmetic in Prolog

Prolog provides a number of basic arithmetic tools for manipulating integers (that is, numbers of the form ...-3, -2, -1, 0, 1, 2, 3, 4...). Most Prolog implementation also provide tools for handling real numbers (or floating point numbers) such as 1.53 or 6.35×10^5, but we're not going to discuss these, for they are not particularly useful for the symbolic processing tasks discussed in this course. Integers, on the other hand, are useful for various tasks (such as finding the length of a list), so it is important to understand how to work with them. We'll start by looking at how Prolog handles the four basic operations of addition, multiplication, subtraction, and division.

Arithmetic examples	Prolog Notation
$6+2=8$	`8 is 6+2.`
$6*2=12$	`12 is 6*2.`
$6-2=4$	`4 is 6-2.`
$6-8=-2$	`-2 is 6-8.`
$6\div 2=3$	`3 is 6/2.`
$7\div 2=3$	`3 is 7/2.`
1 is the remainder when 7 is divided by 2	`1 is mod(7,2).`

(Note that as we are working with integers, division gives us back an integer answer. Thus $7\div 2$ gives 3 as an answer, leaving a reminder of 1.)

Posing the following queries yields the following responses:

```
?- 8 is 6+2.
yes

?- 12 is 6*2.
yes

?- -2 is 6-8.
yes

?- 3 is 6/2.
yes

?- 1 is mod(7,2).
yes
```

More importantly, we can work out the answers to arithmetic questions by using variables. For example:

```
?- X is 6+2.
X = 8

?- X is 6*2.
X = 12

?- R is mod(7,2).
R = 1
```

Moreover, we can use arithmetic operations when we define predicates. Here's a simple example. Let's define a predicate `add_3_and_double`2/ whose arguments are both integers. This predicate takes its first argument, adds three to it, doubles the result, and returns the number obtained as the second argument. We define this predicate as follows:

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
```

And indeed, this works:

```
?- add_3_and_double(1,X).
X = 8

?- add_3_and_double(2,X).
X = 10
```

One other thing. Prolog understands the usual conventions we use for disambiguating arithmetical expressions. For example, when we write $3+2\times 4$ we mean $3+(2\times 4)$ and not $(3+2)\times 4$, and Prolog knows this convention:

```
?- X is 3+2*4.
X = 11
```

5.2 A closer look

That's the basics, but we need to know more. The most important to grasp is this: +, *, -, ÷ and `mod` do *not* carry out any arithmetic. In fact, expressions such as `3+2`, `3-2` and `3*2` are simply *terms*. The functors of these terms are `+`, `-` and `*` respectively, and the arguments are `3` and `2`. Apart from the fact that the functors go between their arguments (instead of in front of them) these are ordinary Prolog terms, and unless we do something special, Prolog will *not* actually do any arithmetic. In particular, if we pose the query

```
?- X = 3+2
```

we *don't* get back the answer `X=5`. Instead we get back

```
X = 3+2
yes
```

That is, Prolog has simply bound the variable `X` to the complex term `3+2`. It has *not* carried out any arithmetic. It has simply done what it usually does: performed unification Similarly, if we pose the query

```
?- 3+2*5 = X
```

we get the response

```
X = 3+2*5
yes
```

Again, Prolog has simply bound the variable `X` to the complex term `3+2*5`. It did *not* evaluate this expression to 13. To force Prolog to actually evaluate arithmetic expressions we have to use the predicate `is` just as we did in our in our earlier examples. In fact, `is` does something very special: it sends a signal to Prolog that says 'Hey! Don't treat this

expression as an ordinary complex term! Call up your built-in arithmetic capabilities and carry out the calculations!'

In short, `is` forces Prolog to act in an unusual way. Normally Prolog is quite happy just unifying variables to structures: that's its job, after all. Arithmetic is something extra that has been bolted on to the basic Prolog engine because it is useful. Unsurprisingly, there are some restrictions on this extra ability, and we need to know what they are.

For a start, the arithmetic expressions to be evaluated must be on the right hand side of `is`. In our earlier examples we carefully posed the query

```
?- X is 6+2.
X = 8
```

which is the right way to do it. If instead we had asked

```
?- 6+2 is X.
```

we would have got an error message saying `instantiation_error`, or something similar.

Moreover, although we are free to use variables on the right hand side of `is`, when we actually carry out evaluation, *the variable must already have been instantiated to an integer*. If the variable is uninstantiated, or if it is instantiated to something other than an integer, we will get some sort of `instantiation_error` message. And this makes perfect sense. Arithmetic *isn't* performed using Prolog usual unification and knowledge base search mechanisms: it's done by calling up a special 'black box' which knows about integer arithmetic. If we hand the black box the wrong kind of data, naturally it's going to complain.

Here's an example. Recall our 'add 3 and double it' predicate.

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
```

When we described this predicate, we carefully said that it added 3 to its first argument, doubled the result, and returned the answer in its second argument. For example, `add_3_and_double(3,X)` returns `X = 12`. We didn't say anything about using this predicate in the reverse direction. For example, we might hope that posing the query

```
?- add_3_and_double(X,12).
```

would return the answer `X=3`. But it doesn't! Instead we get the `instantiation_error` message. Why? Well, when we pose the query this way round, we are asking Prolog to evaluate `12 is (X+3)*2`, which it *can't* do as `X` is not instantiated.

Two final remarks. As we've already mentioned, for Prolog `3 + 2` is just a term. In fact, for Prolog, it really *is* the term *+(3,2)*. The expression `3 + 2` is just a user-friendly notation that's nicer for us to use. This means that if you really want to, you can give Prolog queries like

```
?- X is +(3,2)
```

and Prolog will correctly reply

```
X = 5
```

Actually, you can even given Prolog the query

```
?- is(X,+(3,2))
```

and Prolog will respond

```
X = 5.
```

This is because, for Prolog, the expression `X is +(3,2)` is the term `is(X,+(3,2))`. The expression `X is +(3,2)` is just user friendly notation. Underneath, as always, Prolog is just working away with terms.

Summing up, arithmetic in Prolog is easy to use. Pretty much all you have to remember is to use `is` to force evaluation, that stuff to be evaluated must goes to the right of `is`, and to take care that any variables are correctly instantiated. But there is a deeper lesson that is worth reflecting on. By 'bolting on' the extra capability to do arithmetic we have further widened the distance between the procedural and declarative interpretation of Prolog processing.

5.3 Arithmetic and lists

Probably the most important use of arithmetic in this course is to tell us useful facts about data-structures, such as lists. For example, it can be useful to know how long a list is. We'll give some examples of using lists together with arithmetic capabilities.

How long is a list? Here's a recursive definition.

1. The empty list has length zero.
2. A non-empty list has length 1 + *len*(T), where *len*(T) is the length of its tail.

This definition is practically a Prolog program already. Here's the code we need:

```
len([],0). len([_|T],N) :- len(T,X), N is X+1.
```

This predicate works in the expected way. For example:

```
?- len([a,b,c,d,e,[a,b],g],X).
X = 7
```

Now, this is quite a good program: it's easy to understand and efficient. But there is another method of finding the length of a list. We'll now look at this alternative, because it introduces the idea of **accumulators**, a standard Prolog technique we will be seeing lots more of.

If you're used to other programming languages, you're probably used to the idea of using variables to hold intermediate results. An accumulator is the Prolog analog of this idea.

Here's how to use an accumulator to calculate the length of a list. We shall define a predicate `accLen`3/ which takes the following arguments.

```
accLen(List,Acc,Length)
```

Here `List` is the list whose length we want to find, and `Length` is its length (an integer). What about `Acc`? This is a variable we will use to keep track of intermediate values for length (so it will also be an integer). Here's what we do. When we call this predicate, we are going to give `Acc` an initial value of `0`. We then recursively work our way down the list, adding `1` to `Acc` each time we find a head element, until we reach the empty list. When we do reach the empty set, `Acc` will contain the length of the list. Here's the code:

```
accLen([_|T],A,L) :-  Anew is A+1, accLen(T,Anew,L).

accLen([],A,A).
```

The base case of the definition, unifies the second and third arguments. Why? There are actually *two* reasons. The first is because when we reach the end of the list, the accumulator (the second variable) contains the length of the list. So we give this value (via unification) to the length variable (the third variable). The second is that this trivial unification gives a nice way of stopping the recursion when we reach the empty list. Here's an example trace:

```
?- accLen([a,b,c],0,L).
   Call: (6) accLen([a, b, c], 0, _G449) ?
   Call: (7) _G518 is 0+1 ?
   Exit: (7) 1 is 0+1 ?
   Call: (7) accLen([b, c], 1, _G449) ?
   Call: (8) _G521 is 1+1 ?
   Exit: (8) 2 is 1+1 ?
   Call: (8) accLen([c], 2, _G449) ?
   Call: (9) _G524 is 2+1 ?
   Exit: (9) 3 is 2+1 ?
   Call: (9) accLen([], 3, _G449) ?
   Exit: (9) accLen([], 3, 3) ?
   Exit: (8) accLen([c], 2, 3) ?
   Exit: (7) accLen([b, c], 1, 3) ?
   Exit: (6) accLen([a, b, c], 0, 3) ?
```

As a final step, we'll define a predicate which calls `accLen` for us, and gives it the initial value of 0:

```
leng(List,Length) :- accLen(List,0,Length).
```

So now we can pose queries like this:

```
?- leng([a,b,c,d,e,[a,b],g],X).
```

Accumulators are extremely common in Prolog programs. (We'll see another accumulator based program later in this lecture, and many more in the rest of the course.) But why is this? In what way is `accLen` better than `len`? After all, it looks more difficult. The answer is that `accLen` is **tail recursive** while `len` is not. In tail recursive programs the result is all calculated once we reached the bottom of the recursion and just has to be passed up. In recursive programs which are not tail recursive there are goals in one level of recursion which have to wait for the answer of a lower level of recursion before they can be evaluated. To understand this, compare the traces for the queries `accLen([a,b,c],0,L)` (see above) and `len([a,b,c],0,L)` (given below). In the first case the result is built while going into the recursion – once the bottom is reached at `accLen([],3,_G449)` the result is there and only has to be passed up. In the second case the result is built while coming out of the recursion – the result of `len([b,c], _G481)`, for instance, is only computed after the recursive call of `len` has been completed and the result of `len([c], _G489)` is known.

```
?- len([a,b,c],L).
```

```
Call: (6) len([a, b, c], _G418) ?
Call: (7) len([b, c], _G481) ?
Call: (8) len([c], _G486) ?
Call: (9) len([], _G489) ?
Exit: (9) len([], 0) ?
Call: (9) _G486 is 0+1 ?
Exit: (9) 1 is 0+1 ?
Exit: (8) len([c], 1) ?
Call: (8) _G481 is 1+1 ?
Exit: (8) 2 is 1+1 ?
Exit: (7) len([b, c], 2) ?
Call: (7) _G418 is 2+1 ?
Exit: (7) 3 is 2+1 ?
Exit: (6) len([a, b, c], 3) ?
```

5.4 Comparing integers

Some Prolog arithmetic predicates actually do carry out arithmetic all by themselves (that is, without the assistance of `is`). These are the operators that compare integers.

Arithmetic examples	Prolog Notation
$x < y$	`X < Y.`
$x \leq y$	`X =< Y.`
$x = y$	`X =:= Y.`
$x \neq y$	`X =\= Y.`
$x \geq y$	`X >= Y`
$x > y$	`X > Y`

These operators have the obvious meaning:

```
?- 2 < 4.
yes

?- 2 =< 4.
yes

?- 4 =< 4.
yes

?- 4=:=4.
```

```
yes

?- 4=\=5.
yes

?- 4=\=4.
no

?- 4 >= 4.
yes

?- 4 > 2.
yes
```

Moreover, they force both their right-hand and left-hand arguments to be evaluated:

```
?- 2 < 4+1.
yes

?- 2+1 < 4.
yes

?- 2+1 < 3+2.
yes
```

Note that `=:=` really is different from `=`, as the following examples show:

```
?- 4=4.
yes

?- 2+2 =4.
no

?- 2+2 =:= 4.
yes
```

That is, `=` tries to unify its arguments; it does *not* force arithmetic evaluation. That's `=:=`'s job.

Whenever we use these operators, we have to take care that any variables are instantiated. For example, all the following queries lead to instantiation errors.

```
?- X < 3.

?- 3 < Y.

?- X =:= X.
```

Moreover, variables have to be instantiated to *integers*. The query

```
?- X = 3, X < 4.
```

succeeds. But the query

```
?- X = b, X < 4.
```

fails.

OK, let's now look at an example which puts Prolog's abilities to compare numbers to work. We're going to define a predicate which takes takes a list of non-negative integers as its first argument, and returns the maximum integer in the list as its last argument. Again, we'll use an accumulator. As we work our way down the list, the accumulator will keep track of the highest integer found so far. If we find a higher value, the accumulator will be updated to this new value. When we call the program, we set accumulator to an initial value of 0. Here's the code. Note that there are *two* recursive clauses:

```
accMax([H|T],A,Max) :-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max) :-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).
```

The first clause tests if the head of the list is larger than the largest value found so far. If it is, we set the accumulator to this new value, and then recursively work through the tail of the list. The second clause applies when the head is less than or equal to the accumulator; in this case we recursively work through the tail of the list using the old accumulator value. Finally, the base clause unifies the second and third arguments; it gives the highest value we found while going through the list to the last argument. Here's how it works:

```
accMax([1,0,5,4],0,_5810)

accMax([0,5,4],1,_5810)

accMax([5,4],1,_5810)

accMax([4],5,_5810)

accMax([],5,_5810)

accMax([],5,5)
```

Again, it's nice to define a predicate which calls this, and initializes the accumulator. But wait: what should we initialize the accumulator to? If you say 0, this means you are assuming that all the numbers in the list are positive. But suppose we give a list of negative integers as input. Then we would have

```
?- accMax([-11,-2,-7,-4,-12],0,Max).

Max = 0
yes
```

This is *not* what we want: the biggest number on the list is -2. Our use of 0 as the initial value of the accumulator has ruined everything, because it's bigger than any number on the list.

There's an easy way around this: since our input list will always be a list of integers, simply initialize the accumulator to the head of the list. That way we guarantee that the accumulator is initialized to a number on the list. The following predicate does this for us:

```
max(List,Max) :-
     List = [H|_],
     accMax(List,H,Max).
```

So we can simply say:

```
?- max([1,2,46,53,0],X).

X = 53
yes
```

And furthermore we have:

```
?- max([-11,-2,-7,-4,-12],X).

X = -2
yes
```

6

More Lists

This lecture has two main goals:

1. To define **append**, a predicate for concatenating two lists, and illustrate what can be done with it.
2. To discuss two ways of reversing a list: a naive method using append, and a more efficient method using accumulators.

6.1 Append

We shall define an important predicate `append/3` whose arguments are all lists. Viewed declaratively, `append(L1,L2,L3)` will hold when the list `L3` is the result of concatenating the lists `L1` and `L2` together ('concatenating' means 'joining the lists together, end to end'). For example, if we pose the query

```
?- append([a,b,c],[1,2,3],[a,b,c,1,2,3]).
```

or the query

```
?- append([a,[foo,gibble],c],[1,2,[[],b]],
          [a,[foo,gibble],c,1,2,[1,2,[[],b]]).
```

we will get the response 'yes'. On the other hand, if we pose the query

```
?- append([a,b,c],[1,2,3],[a,b,c,1,2]).
```

or the query

```
?- append([a,b,c],[1,2,3],[1,2,3,a,b,c]).
```

we will get the answer ‘no’.

From a procedural perspective, the most obvious use of `append` is to concatenate two lists together. We can do this simply by using a variable as the third argument: the query

```
?- append([a,b,c],[1,2,3],L3).
```

yields the response

```
L3 = [a,b,c,1,2,3]
yes.
```

But (as we shall soon see) we can also use `append` to split up a list. In fact, `append` is a real workhorse. There’s lots we can do with it, and studying it is a good way to gain a better understanding of list processing in Prolog.

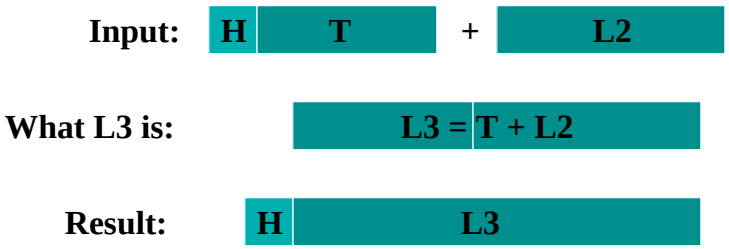
6.1.1 Defining append

Here’s how `append/3` is defined:

```
append([],L,L).
append([H|T],L2,[H|L3]) :- append(T,L2,L3).
```

This is a recursive definition. The base case simply says that appending the empty list to any list whatsoever yields that same list, which is obviously true.

But what about the recursive step? This says that when we concatenate a non-empty list `[H|T]` with a list `L2`, we end up with the list whose head is `H` and whose tail is the result of concatenating `T` with `L2`. It may be useful to think about this definition pictorially:

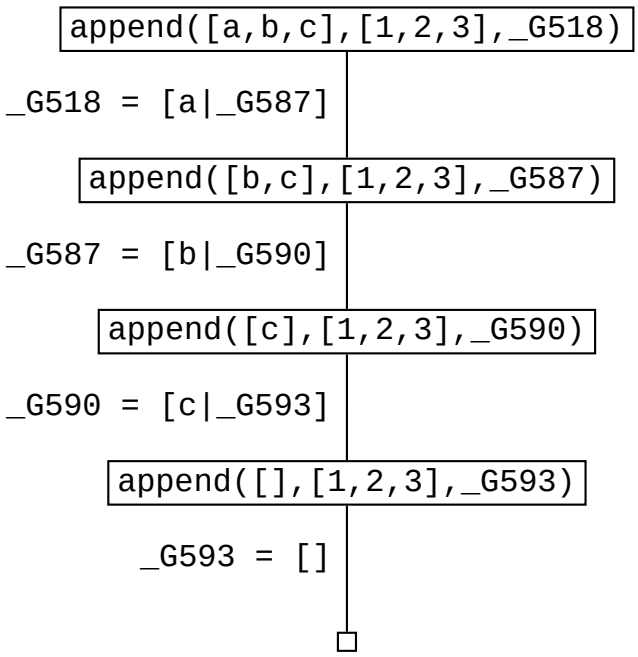


But what is the procedural meaning of this definition? What actually goes on when we use `append` to glue two lists together? Let's take a detailed look at what happens when we pose the query `append([a,b,c],[1,2,3],X)`.

When we pose this query, Prolog will match this query to the head of the recursive rule, generating a new internal variable (say `_G518`) in the process. If we carried out a trace of what happens next, we would get something like the following:

```
append([a, b, c], [1, 2, 3], _G518)
append([b, c], [1, 2, 3], _G587)
append([c], [1, 2, 3], _G590)
append([], [1, 2, 3], _G593)
append([], [1, 2, 3], [1, 2, 3])
append([c], [1, 2, 3], [c, 1, 2, 3])
append([b, c], [1, 2, 3], [b, c, 1, 2, 3])
append([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3])

X = [a, b, c, 1, 2, 3]
yes
```

The basic pattern should be clear: in the first four lines we see that Prolog recurses its way down the list in its first argument until it can apply the base case of the recursive definition. Then, as the next four lines show, it then stepwise 'fills in' the result. How is this 'filling in' process carried out? By successively instantiating the variables `_G593`, `_G590`, `_G587`, and `_G518`. But while it's important to grasp this basic pattern, it doesn't tell us all we need to know about the way `append` works, so let's dig deeper. Here is the search tree for the query `append([a,b,c],[1,2,3],X)` and then we'll work carefully through the steps in the trace, making a careful note of what our goals are, and what the variables are instantiated to. Try to relate this to the search tree.



Goal 1: `append([a,b,c],[1,2,3],_G518)`. Prolog matches this to the head of the recursive rule (that is, `append([H|T],L2,[H|L3])`). Thus `_G518` is matched to `[a|L3]`, and Prolog has the new goal `append([b,c],[1,2,3],L3)`. It generates a new variable `_G587` for `L3`, thus we have that `_G518 = [a|_G587]`.

Goal 2: `append([b,c],[1,2,3],_G587)`. Prolog matches this to the head of the recursive rule, thus `_G587` is matched to `[b|L3]`, and Prolog has the new goal `append([c],[1,2,3],L3)`. It generates the internal variable `_G590` for `L3`, thus we have that `_G587 = [b|_G590]`.

Goal 3: `append([c],[1,2,3],_G590)`. Prolog matches this to the head of the recursive rule, thus `_G590` is matched to `[c|L3]`, and Prolog has the new goal `append([],[1,2,3],L3)`. It generates the internal variable `_G593` for `L3`, thus we have that `_G590 = [c|_G593]`.

Goal 4: `append([],[1,2,3],_G593)`. At last: Prolog can use the base clause (that is, `append([],L,L)`). And in the four successive matching steps, Prolog will obtain answers to Goal 4, Goal 3, Goal 2, and Goal 1. Here's how.

Answer to Goal 4: `append([],[1,2,3],[1,2,3])`. This is because when we match Goal 4 (that is, `append([],[1,2,3],_G593)` to the base clause, `_G593` is matched to `[1,2,3]`.

Answer to Goal 3: `append([c],[1,2,3],[c,1,2,3])`. Why? Because Goal 3 is `append([c],[1,2,3],_G590])`, and `_G590 = [c|_G593]`, and we have just matched `_G593` to `[1,2,3]`. So `_G590` is matched to `[c,1,2,3]`.

Answer to Goal 2: `append([b,c],[1,2,3],[b,c,1,2,3])`. Why? Because Goal 2 is `append([b,c],[1,2,3],_G587])`, and `_G587 = [b|_G590]`, and we have just matched `_G590` to `[c,1,2,3]`. So `_G587` is matched to `[b,c,1,2,3]`.

Answer to Goal 1: `append([a,b,c],[1,2,3],[b,c,1,2,3])`. Why? Because Goal 2 is `append([a,b,c],[1,2,3],_G518])`, `_G518 = [a|_G587]`, and we have just matched `_G587` to `[b,c,1,2,3]`. So `_G518` is matched to `[a,b,c,1,2,3]`.

Thus Prolog now knows how to instantiate `X`, the original query variable. It tells us that `X = [a,b,c,1,2,3]`, which is what we want.

Work through this example carefully, and make sure you fully understand the pattern of variable instantiations, namely:

```
_G518 = [a|_G587]
      = [a|[b|_G590]]
      = [a|[b|[c|_G593]]]
```

For a start, this type of pattern lies at the heart of the way `append` works. Moreover, it illustrates a more general theme: the use of matching to build structure. In a nutshell, the recursive calls to append build up this nested pattern of variables which code up the required answer. When Prolog finally instantiates the innermost variable `_G593` to `[1, 2, 3]`, the answer crystallizes out, like a snowflake forming around a grain of dust. But it is matching, not magic, that produces the result.

6.1.2 Using append

Now that we understand how `append` works, let's see how we can put it to work.

One important use of `append` is to split up a list into two consecutive lists. For example:

```
?- append(X,Y,[a,b,c,d]).

X = []
Y = [a,b,c,d] ;

X = [a]
Y = [b,c,d] ;
```

```
X = [a,b]
Y = [c,d] ;

X = [a,b,c]
Y = [d] ;

X = [a,b,c,d]
Y = [] ;

no
```

That is, we give the list we want to split up (here `[a,b,c,d]`) to `append` as the third argument, and we use variables for the first two arguments. Prolog then searches for ways of instantiating the variables to two lists that concatenate to give the third argument, thus splitting up the list in two. Moreover, as this example shows, by backtracking, Prolog can find all possible ways of splitting up a list into two consecutive lists.

This ability means it is easy to define some useful predicates with `append`. Let's consider some examples. First, we can define a program which finds **prefixes** of lists. For example, the prefixes of `[a,b,c,d]` are `[]`, `[a]`, `[a,b]`, `[a,b,c]`, and `[a,b,c,d]`. With the help of `append` it is straightforward to define a program `prefix/2`, whose arguments are both lists, such that `prefix(P,L)` will hold when `P` is a prefix of `L`. Here's how:

```
prefix(P,L) :- append(P,_,L).
```

This says that list `P` is a prefix of list `L` when there is some list such that `L` is the result of concatenating `P` with that list. (We use the anonymous variable since we don't care what that other list is: we only care that there is some such list or other.) This predicate successfully finds prefixes of lists, and moreover, via backtracking, finds them all:

```
?- prefix(X,[a,b,c,d]).

X = [] ;

X = [a] ;

X = [a,b] ;

X = [a,b,c] ;
```

```
X = [a,b,c,d] ;

no
```

In a similar fashion, we can define a program which finds **suffixes** of lists. For example, the suffixes of `[a,b,c,d]` are `[]`, `[d]`, `[c,d]`, `[b,c,d]`, and `[a,b,c,d]`. Again, using `append` it is easy to define `suffix/2`, a predicate whose arguments are both lists, such that `suffix(S,L)` will hold when `S` is a suffix of `L`:

```
suffix(S,L) :- append(_,S,L).
```

That is, list `S` is a suffix of list `L` if there is some list such that `L` is the result of concatenating that list with `S`. This predicate successfully finds suffixes of lists, and moreover, via backtracking, finds them all:

```
?- suffix(X,[a,b,c,d]).

X = [a,b,c,d] ;

X = [b,c,d] ;

X = [c,d] ;

X = [d] ;

X = [] ;

no
```

Make sure you understand why the results come out in this order.

And now it's very easy to define a program that finds **sublists** of lists. The sublists of `[a,b,c,d]` are `[]`, `[a]`, `[b]`, `[c]`, `[d]`, `[a,b]`, `[b,c]`, `[c,d]`, `[d,e]`, `[a,b,c]`, `[b,c,d]`, and `[a,b,c,d]`. Now, a little thought reveals that the sublists of a list L are simply the *prefixes of suffixes of* L. Think about it pictorially:

And of course, we have both the predicates we need to pin this ideas down: we simply define

```
sublist(SubL,L) :- suffix(S,L),prefix(SubL,S).
```

That is, `SubL` is a sublist of `L` if there is some suffix `S` of `L` of which `SubL` is a prefix. This program doesn't *explicitly* use `append`, but of course, under the surface, that's what's doing the work for us, as both `prefix` and `suffix` are defined using `append`.

6.2 Reversing a list

Append is a useful predicate, and it is important to know how to use it. But it is just as important to know that it can be a source of inefficiency, and that you probably don't want to use it all the time.

Why is `append` a source of inefficiency? If you think about the way it works, you'll notice a weakness: `append` doesn't join two lists in one simple action. Rather, it needs to work its way down its first argument until it finds the end of the list, and only then can it carry out the concatenation.

Now, often this causes no problems. For example, if we have two lists and we just want to concatenate them, it's probably not too bad. Sure, Prolog will need to work down the length of the first list, but if the list is not too long, that's probably not too high a price to pay for the ease of working with `append`.

But matters may be very different if the first two arguments are given as variables. As we've just seen, it can be very useful to give `append` variables in its first two arguments, for this lets Prolog search for ways of splitting up the lists. But there is a price to pay: a lot of searching is going on, and this can lead to very inefficient programs.

To illustrate this, we shall examine the problem of reversing a list. That is, we will examine the problem of defining a predicate which takes a list (say `[a,b,c,d]`) as input and returns a list containing the same elements in the reverse order (here `[d,c,b,a]`).

Now, a `reverse` predicate is a useful predicate to have around. As you will have realized by now, lists in Prolog are far easier to access from the front than from the back. For example, to pull out the head of a list `L`, all we have to do is perform the unification `[H|_] = L`; this results in `H` being instantiated to the head of `L`. But pulling out the last element of an arbitrary list is harder: we can't do it simply using unification. On the other hand, if we had a predicate which reversed lists, we could first reverse the input list, and then pull out the head of the reversed list, as this would give us the last element of the original list. So a `reverse` predicate could be a useful tool. However, as we may have to reverse large lists, we would like this tool to be efficient. So we need to think about the problem carefully.

And that's what we're going to do now. We will define two reverse predicates: a naive one, defined with the help of `append`, and a more efficient (and indeed, more natural) one defined using accumulators.

6.2.1 Naive reverse using append

Here's a recursive definition of what is involved in reversing a list:

1. If we reverse the empty list, we obtain the empty list.
2. If we reverse the list `[H|T]`, we end up with the list obtained by reversing `T` and concatenating with `[H]`.

To see that the recursive clause is correct, consider the list `[a,b,c,d]`. If we reverse the tail of this list we obtain `[d,c,b]`. Concatenating this with `[a]` yields `[d,c,b,a]`, which is the reverse of `[a,b,c,d]`.

With the help of `append` it is easy to turn this recursive definition into Prolog:

```
naiverev([],[]).

naiverev([H|T],R) :- naiverev(T,RevT),append(RevT,[H],R).
```

Now, this definition is correct, but it does an awful lot of work. It is *very* instructive to look at a trace of this program. This shows that the program is spending a lot of time carrying out appends. This shouldn't be too surprising: after, all, we are calling `append` recursively. The result is very inefficient (if you run a trace, you will find that it takes about 90 steps to reverse an eight element list) and hard to understand (the predicate spends most of it time in the recursive calls to `append`, making it very hard to see what is going on).

Not nice. And as we shall now see, there *is* a better way.

6.2.2 Reverse using an accumulator

The better way is to use an accumulator. The underlying idea is simple and natural. Our accumulator will be a list, and when we start it will be empty. Suppose we want to reverse `[a,b,c,d]`. At the start, our accumulator will be `[]`. So we simply take the head of the list we are trying to reverse and add it as the head of the accumulator. We then carry on processing the tail, thus we are faced with the task of reversing `[b,c,d]`, and our accumulator is `[a]`. Again we take the head of the list we are trying to reverse and add it as the head of the accumulator (thus our new accumulator is [b,a]) and carry on trying to reverse `[c,d]`. Again we use the same idea, so we get a new accumulator `[c,b,a]`, and try to reverse `[d]`.

Needless to say, the next step yields an accumulator `[d,c,b,a]` and the new goal of trying to reverse `[]`. This is where the process stops: *and our accumulator contains the reversed list we want*. To summarize: the idea is simply to work our way through the list we want to reverse, and push each element in turn onto the head of the accumulator, like this:

```
List: [a,b,c,d]  Accumulator: []
List: [b,c,d]    Accumulator: [a]
List: [c,d]      Accumulator: [b,a]
List: [d]        Accumulator: [c,b,a]
List: []         Accumulator: [d,c,b,a]
```

This will be efficient because we simply blast our way through the list once: we don't have to waste time carrying out concatenation or other irrelevant work.

It's also easy to put this idea in Prolog. Here's the accumulator code:

```
accRev([],A,A).

accRev([H|T],A,R) :- accRev(T,[H|A],R).
```

This is classic accumulator code: it follows the same pattern as the arithmetic examples we examined in the previous lecture. The recursive clause is responsible for chopping off the head of the input list, and pushing it onto the accumulator. The base case halts the program, and copies the accumulator to the final argument.

As is usual with accumulator code, it's a good idea to write a predicate which carries out the required initialization of the accumulator for us:

```
rev(L,R) :- accRev(L,[],R).
```

Again, it is instructive to run some traces on this program and compare it with `naiverev`. The accumulator based version is *clearly* better. For example, it takes about 20 steps to reverse an eight element list, as opposed to 90 for the naive version. Moreover, the trace is far easier to follow. The idea underlying the accumulator based version is simpler and more natural than the recursive calls to `append`.

Summing up, `append` is a useful program, and you certainly should not be scared of using it. However, you also need to be aware that it is a source of inefficiency, so when you use it, ask yourself whether there is a better way. And often there is. The use of accumulators is often better, and (as the `reverse` example show) accumulators can be a natural way of handling list processing tasks. Moreover, as we shall learn later in the course, there are more sophisticated ways of thinking about lists (namely by viewing them as *difference lists*) which can also lead to dramatic improvements in performance.

7

Definite Clause Grammars

This lecture has two main goals:

1. To introduce context free grammars (CFGs) and some related concepts.
2. To introduce definite clause grammars (DCGs), an built-in Prolog mechanism for working with context free grammars (and other kinds of grammar too).

7.1 Context free grammars

Prolog has been used for many purposes, but its inventor, Alain Colmerauer, was a computational linguist, and computational linguistics remains a classic application for the language. Moreover, Prolog offers a number of tools which make life easier for computational linguists, and today we are going to start learning about one of the most useful of these: *Definite Clause Grammars*, or DCGs as they are usually called.

DCGs are a special notation for defining **grammars**. So, before we go any further, we'd better learn what a grammar is. We shall do so by discussing **context free grammars** (or CFGs). The basic idea of context free grammars is simple to understand, but don't be fooled into thinking that CFGs are toys. They're not. While CFGs aren't powerful enough to cope with the syntactic structure of all natural languages (that is, the kind of languages that human beings use), they can certainly handle most aspects of the syntax of many natural languages (for example, English, German, and French) in a reasonably natural way.

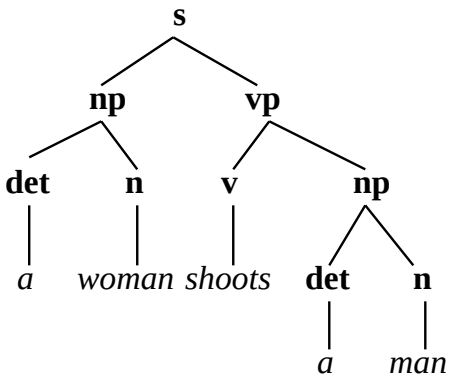
So what is a context free grammar? In essence, a finite collection of rules which tell us that certain sentences are grammatical (that is, syntactically correct) and what their grammatical structure actually is. Here's a simple context free grammar for a small fragment of English:

```
s -> np vp
np -> det n
vp -> v np
vp -> v
det -> a
det -> the
n -> woman
n -> man
v -> shoots
```

What are the ingredients of this little grammar? Well, first note that it contains three types of symbol. There's `->`, which is used to define the rules. Then there are the symbols written like this: `s`, `np`, `vp`, `det`, `n`, `v`. These symbols are called **non-terminal symbols**; we'll soon learn why. Each of these symbols has a traditional meaning in linguistics: `s` is short for **sentence**, `np` is short for **noun phrase**, `vp` is short for **verb phrase**, and `det` is short for **determiner**. That is, each of these symbols is shorthand for a **grammatical category**. Finally there are the symbols in italics: *a*, *the*, *woman*, *man*, and *shoots*. A computer scientist would probably call these **terminal symbols** (or: the **alphabet**), and linguists would probably call them **lexical items**. We'll use these terms occasionally, but often we'll make life easy for ourselves and just call them words.

Now, this grammar contains nine **rules**. A context free rule consists of a single non-terminal symbol, followed by `->`, followed by a finite sequence made up of terminal and/or non-terminal symbols. All nine items listed above have this form, so they are all legitimate context free rules. What do these rules mean? They tell us how different grammatical categories can be built up. Read `->` as *can consist of*, or *can be built out of*. For example, the first rule tells us that a sentence can consist of a noun phrase followed by a verb phrase. The third rule tells us that a verb phrase can consist of a verb followed by a noun phrase, while the fourth rule tells us that there is another way to build a verb phrase: simply use a verb. The last five rules tell us that *a* and *the* are determiners, that *man* and *woman* are nouns, and that *shoots* is a verb.

Now, consider the string of words *a woman shoots a man*. Is this grammatical according to our little grammar? And if it is, what structure does it have? The following tree answers both questions:



Right at the top we have a node marked `s`. This node has two daughters, one marked `np`, and one marked `vp`. Note that this part of the diagram agrees with the first rule of the grammar, which says that an `s` can be built out of an `np` and a `vp`. (A linguist would say that this part of the tree is **licensed** by the first rule.) In fact, as you can see, *every* part of the tree is licensed by one of our rules. For example, the two nodes marked `np` are licensed by the rule that says that an `np` can consist of a `det` followed by an `n`. And, right at the bottom of the diagram, all the words in *a woman shoots a man* are licensed by a rule. Incidentally, note that the terminal symbols only decorate the nodes right at the bottom of the tree (the **terminal nodes**) while non-terminal symbols only decorate nodes that are higher up in the tree (the **non-terminal nodes**).

Such a tree is called a **parse tree**, and it gives us two sorts of information: information about **strings** and information about **structure**. This is an important distinction to grasp, so let's have a closer look, and learn some important terminology while we are doing so.

First, if we are given a string of words, and a grammar, and it turns out that we *can* build a parse tree like the one above (that is, a tree that has `s` at the top node, and every node in the tree is licensed by the grammar, and the string of words we were given is listed in the correct order along the terminal nodes) then we say that the string is **grammatical** (according to the given grammar). For example, the string *a woman shoots a man* is grammatical according to our little grammar (and indeed, any reasonable grammar of English would classify it as grammatical). On the other hand, if there isn't any such tree, the string is **ungrammatical** (according to the given grammar). For example, the string *woman a woman man a shoots* is ungrammatical according to our little grammar (and any reasonable grammar of English would classify it as ungrammatical). The **language generated by a grammar** consists of all the strings that the grammar classifies as grammatical. For example, *a woman shoots a man* also belongs to the language generated by our little grammar, and so does *a man shoots the woman*. A context free **recognizer** is a program which correctly tells us whether or not a string belongs to the language generated by a context free grammar. To put it another way, a recognizer is a program that correctly classifies strings as grammatical or ungrammatical (relative to some grammar).

But often, in both linguistics and computer science, we are not merely interested in whether a string is grammatical or not, we want to know *why* it is grammatical. More precisely, we often want to know what its structure is, and this is exactly the information a parse tree gives us. For example, the above parse tree shows us how the words in *a woman shoots a man* fit together, piece by piece, to form the sentence. This kind of information would be important if we were using this sentence in some application and needed to say what it actually meant (that is, if we wanted to do **semantics**). A context free **parser** is a program which correctly decides whether a string belongs to the language generated by a context free grammar *and also tells us what its structure is*. That is, whereas a recognizer merely says 'Yes, grammatical' or 'No, ungrammatical' to each string, a parser actually builds the associated parse tree and gives it to us.

It remains to explain one final concept, namely what a **context free language** is. (Don't get confused: we've told you what a context free *grammar* is, but not what a context free *language* is.) Quite simply, a context free language is a language that can be generated by a context free grammar. Some languages are context free, and some are not. For example, it seems plausible that English is a context free language. That is, it is probably possible to write a context free grammar that generates all (and only) the sentences that native speakers find acceptable. On the other hand, some dialects of Swiss-German are *not* context free. It can be proved mathematically that no context free grammar can generate all (and only) the sentences that native speakers find acceptable. So if you wanted to write a grammar for such dialects, you would have to employ additional grammatical mechanisms, not merely context free rules.

7.1.1 CFG recognition using append

That's the theory, but how do we work with context free grammars in Prolog? To make things concrete: suppose we are given a context free grammar. How can we write a recognizer for it? And how can we write a parser for it? This week we'll look at the first question in detail. We'll first show how (rather naive) recognizers can be written in Prolog, and then show how more sophisticated recognizers can be written with the help of difference lists. This discussion will lead us to definite clause grammars, Prolog's built-in grammar tool. Next week we'll look at definite clause grammars in more detail, and learn (among other things) how to use them to define parsers.

So: given a context free grammar, how do we define a recognizer in Prolog? In fact, Prolog offers a very direct answer to this question: we can simply write down Prolog clauses that correspond, in an obvious way, to the grammar rules. That is, we can simply 'turn the grammar into Prolog'.

Here's a simple (though as we shall learn, inefficient) way of doing this. We shall use lists to represent strings. For example, the string *a woman shoots a man* will be represented by

the list `[a,woman,shoots,a,man]`. Now, we have already said that the `->` symbol used in context free grammars means *can consist of*, or *can be built out of*, and this idea is easily modeled using lists. For example, the rule `s -> np vp` can be thought of as saying: *a list of words is an* `s` *list* if it is the result of concatenating an `np` *list with a* `vp` *list*. As we know how to concatenate lists in Prolog (we can use `append`), it should be easy to turn these kinds of rules into Prolog. And what about the rules that tell us about individual words? Even easier: we can simply view `n ->` *woman* as saying that the list `[woman]` is an `n` list.

If we turn these ideas into Prolog, this is what we get:

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

np(Z) :- det(X), n(Y), append(X,Y,Z).

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

vp(Z) :-  v(Z).

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

n([woman]).
n([man]).

v([shoots]).
```

The correspondence between the CFG rules and the Prolog should be clear. And to use this program as a recognizer, we simply pose the obvious queries. For example:

```
?- s([a,woman,shoots,a,man]).
yes
```

In fact, because this is a simple declarative Prolog program, we can do more than this: we can also **generate** all the sentences this grammar produces. In fact, our little grammar generates 20 sentences. Here are the first five:

```
?- s(X).

X = [the,woman,shoots,the,woman] ;

X = [the,woman,shoots,the,man] ;
```

```
X = [the,woman,shoots,a,woman] ;

X = [the,woman,shoots,a,man] ;

X = [the,woman,shoots]
```

Moreover, we're not restricted to posing questions about sentences: we can ask about other grammatical categories. For example:

```
?- np([a,woman]).
yes
```

And we can generate noun phrases with the following query.

```
?- np(X).
```

Now this is rather nice. We have a simple, easy to understand program which corresponds with our CFG in an obvious way. Moreover, if we added more rules to our CFG, it would be easy to alter the program to cope with the new rules.

But there is a problem: the program doesn't use the input sentence to guide the search. Make a trace for the query `s([a,man,shoots])` and you will see that the program 'guesses' noun phrases and verb phrases and then afterwards checks whether these can be combined to form the sentence `[a,man,shoots]`. Prolog will find that `[the,woman]` is a noun phrase and `[shoots,the,woman]` a verb phrase and then it will check whether concatenating these two lists happens to yield `[a,man,shoots]`, which of course fails. So, Prolog starts to backtrack and the next thing it will try is whether concatenating the noun phrase `[the,woman]` and the verb phrase `[shoots,the,man]` happens to yield `[a,man,shoots]`. It will go on like this until it finally produces the noun phrase `[the,man]` and the verb phrase `[shoots]`. The problem obviously is, that the goals `np(X)` and `vp(Y)` are called with uninstantiated variables as arguments.

So, how about changing the rules in such a way that `append` becomes the first goal:

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

np(Z) :- append(X,Y,Z), det(X), n(Y).

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

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

```
det([the]).
det([a]).

n([woman]).
n([man]).

v([shoots]).
```

Now, we first use append to split up the input list. This instantiates the varibales `X` and `Y`, so that the other goals are all called with instantiated arguments. However, the program is still not perfect: it uses `append` a lot and, even worse, it uses `append` with uninstantiated variables in the first two arguments. We saw in the previous chapter that that is a source of inefficiency. And indeed, the performance of this recognizer is very bad. It is revealing to trace through what actually happens when this program analyses a sentence such as *a woman shoots a man*. As you will see, relatively few of the steps are devoted to the real task of recognizing the sentences: most are devoted to using `append` to decompose lists. This isn't much of a problem for our little grammar, but it certainly would be if we were working with a more realistic grammar capable of generating a large number of sentences. We need to do something about this.

7.1.2 CFG recognition using difference lists

A more efficient implementation can be obtained by making use of *difference lists*. This is a sophisticated (and, once you've understood it, beautiful) Prolog technique that can be used for a variety of purposes. We won't discuss the idea of difference lists in any depth: we'll simply show how they can be used to rewrite our recognizer more efficiently.

The key idea underlying difference lists is to represent the information about grammatical categories not as a single list, but as the difference between two lists. For example, instead of representing *a woman shoots a man* as `[a,woman,shoots,a,man]` we might represent it as the pair of lists

```
[a,woman,shoots,a,man] [].
```

Think of the first list as *what needs to be consumed* (or if you prefer: the *input list*), and the second list as *what we should leave behind* (or: the *output list*). Viewed from this (rather procedural) perspective the difference list

```
[a,woman,shoots,a,man] [].
```

represents the sentence *a woman shoots a man* because it says: *If I consume all the symbols on the left, and leave behind the symbols on the right, I have the sentence I am interested in.*

That is: the sentence we are interested in is the difference between the contents of these two lists.

Difference representations are not unique. In fact, we could represent *a woman shoots a man* in infinitely many ways. For example, we could also represent it as

```
[a,woman,shoots,a,man,ploggle,woggle]  [ploggle,woggle].
```

Again the point is: if we consume all the symbols on the left, and leave behind the symbols on the right, we have the sentence we are interested in.

That's all we need to know about difference lists to rewrite our recognizer. If we bear the idea of 'consuming something, and leaving something behind' in mind', we obtain the following recognizer:

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

np(X,Z) :- det(X,Y), n(Y,Z).

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

vp(X,Z) :-  v(X,Z).

det([the|W],W).
det([a|W],W).

n([woman|W],W).
n([man|W],W).

v([shoots|W],W).
```

The `s` rule says: *I know that the pair of lists* `X` *and* `Z` *represents a sentence if (1) I can consume* `X` *and leave behind a* `Y`*, and the pair* `X` *and* `Y` *represents a noun phrase, and (2) I can then go on to consume* `Y` *leaving Z behind, and the pair* `Y Z` *represents a verb phrase.*

The idea underlying the way we handle the words is similar. The code

```
n([man|W],W).
```

means we are handling *man* as the difference between `[man|W]` and `W`. Intuitively, the difference between what I consume and what I leave behind is precisely the word `man`.

Now, at first this is probably harder to grasp than our previous recognizer. But we have gained something important: *we haven't used* `append`. In the difference list based recognizer, they simply aren't needed, and as we shall see, this makes a *big* difference.

How do we use such grammars? Here's how to recognize sentences:

```
?- s([a,woman,shoots,a,man],[]).
yes
```

This asks whether we can get an `s` by consuming the symbols in `[a,woman,shoots,a,man]`, leaving nothing behind.

Similarly, to generate all the sentences in the grammar, we ask

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

This asks: what values can you give to `X`, such that we get an `s` by consuming the symbols in `X`, leaving nothing behind?

The queries for other grammatical categories also work the same way. For example, to find out if *a woman* is a noun phrase we ask

```
?- np([a,woman],[]).
```

And we generate all the noun phrases in the grammar as follows:

```
?- np(X,[]).
```

You should trace what happens when this program analyses a sentence such as *a woman shoots a man*. As you will see, it is a lot more efficient than our `append` based program. Moreover, as no use is made of `append`, the trace is a lot easier to grasp. So we have made a big step forward.

On the other hand, it has to be admitted that the second recognizer is not as easy to understand, at least at first, and it's a pain having to keep track of all those difference list variables. If only it were possible to have a recognizer as simple as the first and as efficient as the second. And in fact, it *is* possible: this is where DCGs come in.

7.2 Definite clause grammars

So, what are DCGs? Quite simply, *a nice notation for writing grammars that hides the underlying difference list variables*. Let's look at three examples.

7.2.1 A first example

As our first example, here's our little grammar written as a DCG:

```
s --> np,vp.

np --> det,n.

vp --> v,np.
vp --> v.

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

n --> [woman].
n --> [man].

v --> [shoots].
```

The link with the original context free grammar should be utterly clear: this is definitely the most user friendly notation we have used yet. But how do we use this DCG? In fact, we use it in *exactly* the same way as we used our difference list recognizer. For example, to find out whether *a woman shoots a man* is a sentence, we pose the query:

```
?- s([a,woman,shoots,a,man],[]).
```

That is, just as in the difference list recognizer, we ask whether we can get an `s` by consuming the symbols in `[a,woman,shoots,a,man]`, leaving nothing behind.

Similarly, to generate all the sentences in the grammar, we pose the query:

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

This asks what values we can give to `X`, such that we get an `s` by consuming the symbols in `X`, leaving nothing behind.

Moreover, the queries for other grammatical categories also work the same way. For example, to find out if *a woman* is a noun phrase we pose the query:

```
?- np([a,woman],[]).
```

And we generate all the noun phrases in the grammar as follows:

```
?- np(X,[]).
```

What's going on? Quite simply, this DCG *is* our difference list recognizer! That is, DCG notation is essentially syntactic sugar: user friendly notation that lets us write grammars in a natural way. But Prolog translates this notation into the kinds of difference lists discussed before. So we have the best of both worlds: a nice simple notation for working with, *and* the efficiency of difference lists.

There is an easy way to actually see what Prolog translates DCG rules into. Suppose you are working with this DCG (that is, Prolog has already consulted the rules). Then if you pose the query:

```
?- listing(s)
```

you will get the response

```
s(A,B) :-
    np(A,C),
    vp(C,B).
```

This is what Prolog has translated `s -> np,vp` into. Note that (apart from the choice of variables) this is exactly the difference list rule we used in our second recognizer.

Similarly, if you pose the query

```
?- listing(np)
```

you will get

```
np(A,B) :-
    det(A,C),
    n(C,B).
```

This is what Prolog has translated `np -> det,n` into. Again (apart from the choice of variables) this is the difference list rule we used in our second recognizer.

To get a complete listing of the translations of all the rules, simply type

```
?- listing.
```

There is one thing you may observe. Some Prolog implementations translate rules such as

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

not into

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

which was the form we used in our difference list recognizer, but into

```
det(A,B) :-
    'C'(A,the,B).
```

Although the notation is different, the idea is the same. Basically, this says you can get a `B` from an `A` by consuming a `the`. (Note that `'C'` is an atom. So, the goal `'C'(A, the, B)` is just a normal call of the built-in predicate `'C'/3`.)

7.2.2 Adding recursive rules

Our original context free grammar generated only 20 sentences. However it is easy to write context free grammars that generate infinitely many sentences: we need simply use recursive rules. Here's an example. Let's add the following rules to our little grammar:

```
s -> s conj s
conj -> and
conj -> or
conj -> but
```

This rule allows us to join as many sentences together as we like using the words *and*, *but* and *or*. So this grammar classifies sentences such as *The woman shoots the man or the man shoots the woman* as grammatical.

It is easy to turn this grammar into DCG rules. In fact, we just need to add the rules

```
s --> s,conj,s.

conj --> [and].
conj --> [or].
conj --> [but].
```

But what does Prolog do with a DCG like this? Let's have a look.

First, let's add the rules at the *beginning* of the knowledge base before the rule `s --> np,vp`. What happens if we then pose the query `s([a,woman,shoots],[])`? Prolog gets into an infinte loop.

Can you see why? The point is this. Prolog translates DCG rules into ordinary Prolog rules. If we place the recursive rule `s --> s,conj,s` in the knowledge base before the non-recursive rule `s --> np,vp` then the knowledge base will contain the following two Prolog rules, in this order:

```
s(A, B) :-
        s(A, C),
        conj(C, D),
        s(D, B).

s(A, B) :-
        np(A, C),
        vp(C, B).
```

Now, from a declarative perspective this is fine, but from a procedural perspective this is fatal. When it tries to use the first rule, Prolog immediately encounters the goal `s(A,C)`, which it then tries to satisfy using the first rule, whereupon it immediately encounters the goal `s(A, C)`, which it then tries to satisfy using the first rule, whereupon it immediately encounters the goal `s(A, C)`... In short, it goes into infinite loop and does no useful work.

Second, let's add the recursive rule `s --> s,conj,s` at the end of the knowledge base, so that Prolog always ecounters the translation of the non-recursive rule first. What happens now, when we pose the query `s([a,woman,shoots],[])`? Well, Prolog seems to be able to handle it and gives an anwer. But what happens when we pose the query `s([woman,shoot],[])`, i.e. an ungrammatical sentence that is not accepted by our grammar? Prolog again gets into an infinite loop. Since, it is impossible to recognize `[woman,shoot]` as a sentence consisting of a noun phrase and a verb phrase, Prolog tries to analyse it with the rule `s --> s,conj,s` and ends up in the same loop as before.

Notice, that we are having the same problems that we had when we were changing the order of the rules and goals in the definition of `descend` in the chapter on recursion. In that case, the trick was to change the goals of the recursive rule so that the recursive goal was not the first one in the body of the rule. In the case of our recursive DCG, however, this is not a possible solution. Since the order of the goals determines the order of the words in the sentence, we cannot change it just like that. It does make a difference, for example, whether our grammar accepts *the woman shoots the man and the man shoots the woman* (`s --> s,conj,s`) or whether it accepts *and the woman shoots the man the man shoots the woman* (`s --> conj,s,s`).

So, by just reordering clauses or goals, we won't solve the problem. The only possible solution is to introduce a new nonterminal symbol. We could for example use the category `simple_s` for sentences without embedded sentences. Our grammar would then look like this:

```
s --> simple_s.
s --> simple_s conj s.
simple_s --> np,vp.
np --> det,n.
vp --> v,np.
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
conj --> [and].
conj --> [or].
conj --> [but].
```

Make sure that you understand why Prolog doesn't get into infinite loops with this grammar as it did with the previous version.

The moral is: DCGs aren't magic. They are a nice notation, but you can't always expect just to 'write down the grammar as a DCG' and have it work. DCG rules are really ordinary Prolog rules in disguise, and this means that you must pay attention to what your Prolog interpreter does with them.

7.2.3 A DCG for a simple formal language

As our last example, we shall define a DCG for the formal language $a^n b^n$. What is this language? And what is a formal language anyway?

A formal language is simply a set of strings. The term 'formal language' is intended to contrast with the term 'natural language': whereas natural languages are languages that human beings actually use, fomal languages are mathematical objects that computer scientists, logicians, and mathematicians define and study for various purpose.

A simple example of a formal language is $a^n b^n$. There are only two 'words' in this language: the symbol a and the symbol b. The language $a^n b^n$ consist of all strings made up from these two symbols that have the following form: the string must consist of an unbroken block of as of length n, followed by an unbroken block of bs of length n, and

nothing else. So the strings *ab*, *aabb*, *aaabbb* and *aaaabbbb* all belong to $a^n b^n$. (Note that the **empty string** belongs to $a^n b^n$ too: after all, the empty string consists of a block of *a*s of length zero followed by a block of *b*s of length zero.) On the other hand, *aaabb* and *aaabbba* do not belong to $a^n b^n$.

Now, it is easy to write a context free grammar that generates this language:

```
s -> ε
s -> l s r
l -> a
r -> b
```

The first rule says that an *s* can be realized as nothing at all. The second rule says that an *s* can be made up of an *l* (for left) element, followed by an *s*, followed by an *r* (for right) element. The last two rules say that *l* elements and *r* elements can be realized as *a*s and *b*s respectively. It should be clear that this grammar really does generate all and only the elements of $a^n b^n$, including the empty string.

Moreover, it is trivial to turn this grammar into DCG. We can do so as follows:

```
s --> [].
s --> l,s,r.

l --> [a].
r --> [b].
```

And this DCG works exactly as we would hope. For example, to the query

```
?- s([a,a,a,b,b,b],[]).
```

we get the answer ‘yes’, while to the query

```
?- s([a,a,a,b,b,b,b],[]).
```

we get the answer ‘no’. And the query

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

enumerates the strings in the language, starting from `[]`.

8

More Definite Clause Grammars

This lecture has two main goals:

1. To examine two important capabilities offered by DCG notation: **extra arguments** and **extra tests**.
2. To discuss the status and limitations of DCGs.

8.1 Extra arguments

In the previous lecture we only scratched the surface of DCG notation: it actually offers a lot more than we've seen so far. For a start, DCGs allow us to specify extra arguments. Extra arguments can be used for many purposes; we'll examine three.

8.1.1 Context free grammars with features

As a first example, let's see how extra arguments can be used to add *features* to context-free grammars.

Here's the DCG we worked with last week:

```
s --> np,vp.

np --> det,n.

vp --> v,np.
vp --> v.

det --> [the].
```

```
det --> [a].

n --> [woman].
n --> [man].

v --> [shoots].
```

Suppose we wanted to deal with sentences like “She shoots him”, and “He shoots her”. What should we do? Well, obviously we should add rules saying that “he”, “she”, “him”, and “her” are pronouns:

```
pro --> [he].
pro --> [she].
pro --> [him].
pro --> [her].
```

Furthermore, we should add a rule saying that noun phrases can be pronouns:

```
np--> pro.
```

Up to a point, this new DCG works. For example:

```
?- s([she,shoots,him],[]).
yes
```

But there’s an obvious problem. The DCG will also accept a lot of sentences that are clearly wrong, such as “A woman shoots she”, “Her shoots a man”, and “Her shoots she”:

```
?- s([a,woman,shoots,she],[]).
yes

?- s([her,shoots,a,man],[]).
yes

?- s([her,shoots,she],[]).
yes
```

That is, the grammar doesn’t know that “she” and “he” are *subject* pronouns and cannot be used in *object* position; thus “A woman shoots she” is bad because it violates this basic fact about English. Moreover, the grammar doesn’t know that “her” and “him” are *object*

pronouns and cannot be used in *subject* position; thus “Her shoots a man” is bad because it violates this constraint. As for “Her shoots she”, this manages to get both matters wrong at once.

Now, it’s pretty obvious *what* we have to do to put this right: we need to extend the DCG with information about which pronouns can occur in subject position and which in object position. The interesting question: *how* exactly are we to do this? First let’s look at a naive way of correcting this, namely adding new rules:

```
s --> np_subject,vp.

np_subject --> det,n.
np_object  --> det,n.
np_subject --> pro_subject.
np_object  --> pro_object.

vp --> v,np_object.
vp --> v.

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

n --> [woman].
n --> [man].

pro_subject --> [he].
pro_subject --> [she].
pro_object --> [him].
pro_object --> [her].

v --> [shoots].
```

Now this solution “works”. For example,

```
?- s([her,shoots,she],[]).
no
```

But neither computer scientists nor linguists would consider this a good solution. The trouble is, a small addition to the lexicon has led to quite a big change in the DCG. Let’s face it: “she” and “her” (and “he” and “him”) are the same in a lot of respects. But to deal with the property in which they differ (namely, in which position in the sentence they can

occur) we've had to make big changes to the grammar: in particular, we've doubled the number of noun phrase rules. If we had to make further changes (for example, to cope with plural noun phrases) things would get even worse. What we really need is a more delicate programming mechanism that allows us to cope with such facts without being forced to add rules all the time. And here's where the extra arguments come into play. Look at the following grammar:

```
s --> np(subject),vp.

np(_) --> det,n.
np(X) --> pro(X).

vp --> v,np(object).
vp --> v.

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

n --> [woman].
n --> [man].

pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].

v --> [shoots].
```

The key thing to note is that *this new grammar contains no new rules*. It is exactly the same as the first grammar that we wrote above, except that the symbol `np` is associated with a new argument, either `(subject)`, `(object)`, `(_)` and `(X)`. A linguist would say that we've added a **feature** to distinguish various kinds of noun phrase. In particular, note the four rules for the pronouns. Here we've used the extra argument to state which pronouns can occur in subject position, and which occur in object position. Thus these rules are the most fundamental, for they give us the basic facts about how these pronouns can be used.

So what do the other rules do? Well, intuitively, the rule

```
np(X) --> pro(X).
```

uses the extra argument (the variable `X`) to pass these basic facts about pronouns up to noun phrases built out of them: because the variable `X` is used as the extra argument for both

the np and the pronoun, Prolog unification will guarantee that they will be given the same value. In particular, if the pronoun we use is "she" (in which case `X=subject`), then the np wil, through its extra argument (`X=subject`), also be marked as being a subject np. On the other hand, if the pronoun we use is "her" (in which case `X=object`), then the extra argument np will be marked `X=object` too. And this, of course, is exactly the behaviour we want.

On the other hand, although noun phrases built using the rule

```
np(_) --> det,n.
```

also have an extra argument, we've used the anonymous variable as its value. Essentially this means *can be either*, which is correct, for expressions built using this rule (such as "the man" and "a woman") can be used in both subject and object position.

Now consider the rule

```
vp --> v,np(object).
```

This says that to apply this rule we need to use an noun phrase whose extra argument unifies with `object`. This can be *either* noun phrases built from object pronouns *or* noun phrases such as "the man" and "a woman" which have the anonymous variable as the value of the extra argument. Crucially, pronouns marked has having `subject` as the value of the extra argument *can't* be used here: the atoms `object` and `subject` don't unify. Note that the rule

```
s --> np(subject),vp.
```

works in an analogous fashion to prevent noun phrases made of object pronouns from ending up in subject position.

This works. You can check it out by posing the query:

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

As you step through the responses, you'll see that only acceptable English is generated.

But while the intuitive explanation just given is correct, what's *really* going on? The key thing to remember is that DCG rules are really are just a convenient abbreviation. For example, the rule

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

is really syntactic sugar for

```
s(A,B) :-
    np(A,C),
    vp(C,B).
```

That is, as we learned in the previous lecture, the DCG notation is a way of hiding the two arguments responsible for the difference list representation, so that we don't have to think about them. We work with the nice user friendly notation, and Prolog translates it into the clauses just given.

Ok, so we obviously need to ask what

```
s --> np(subject),vp.
```

translates into. Here's the answer:

```
s(A,B) :-
    np(subject,A,C),
    vp(C,B).
```

As should now be clear, the name "extra argument" is a good one: as this translation makes clear, the `(subject)` symbol really *is* just one more argument in an ordinary Prolog rule! Similarly, our noun phrase DCG rules translate into

```
np(A,B,C) :-
    det(B,D),
    n(D,C).
np(A,B,C) :-
    pro(A,B,C).
```

Note that both rules have *three* arguments. The first, `A`, is the extra argument, and the last two are the ordinary, hidden DCG arguments (the two hidden arguments are always the last two arguments).

Incidentally, how do you think we would use the grammar to list the grammatical noun phrases? Well, if we had been working with the DCG rule `np -> det,n` (that is, a rule with no extra arguments) we would have made the query

```
?- np(NP,[]).
```

So it's not too surprising that we need to pose the query

```
?- np(X,NP,[]).
```

when working with our new DCG. Here's what the response would be.

```
X = _2625
NP = [the,woman] ;

X = _2625
NP = [the,man] ;

X = _2625
NP = [a,woman] ;

X = _2625
NP = [a,man] ;

X = subject
NP = [he] ;

X = subject
NP = [she] ;

X = object
NP = [him] ;

X = object
NP = [her] ;

no
```

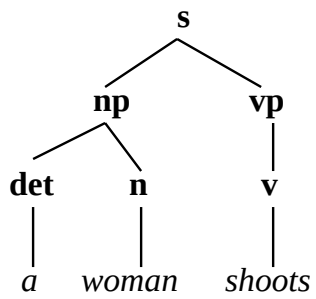
One final remark: *don't be misled by this simplicity of our example*. Extra arguments can be used to cope with some complex syntactic problems. DCGs are no longer the state-of-art grammar development tools they once were, but they're not toys either. Once you know about writing DCGs with extra arguments, you can write some fairly sophisticated grammars.

8.1.2 Building parse trees

So far, the programs we have discussed have been able to *recognize* grammatical structure (that is, they could correctly answer "yes" or "no" when asked whether the input was a

sentence, a noun phrase, and so on) and to *generate* grammatical output. This is pleasant, but we would also like to be able to *parse*. That is, we would like our programs not only to tell us *which* sentences are grammatical, but also to give us an analysis of their structure. In particular, we would like to see the trees the grammar assigns to sentences.

Well, using only standard Prolog tool we can't actually draw nice pictures of trees, but we *can* build data structures which describe trees in a clear way. For example, corresponding to the tree



we could have the following term:

```
s(np(det(a),n(woman)),vp(v(shoots))).
```

Sure: it doesn't *look* as nice, but all the information in the picture is there. And, with the aid of a decent graphics package, it would be easy to turn this term into a picture.

But how do we get DCGs to build such terms? Actually, it's pretty easy. After all, in effect a DCG has to work out what the tree structure is when recognizing a sentence. So we just need to find a way of keeping track of the structure that the DCG finds. We do this by adding extra arguments. Here's how:

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

np(np(DET,N)) --> det(DET),n(N).

vp(vp(V,NP)) --> v(V),np(NP).
vp(vp(V))    --> v(V).

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

n(n(woman)) --> [woman].
n(n(man))   --> [man].

v(v(shoots)) --> [shoots].
```

What's going on here? Essentially we are building the parse trees for the syntactic categories on the left-hand side of the rules out of the parse trees for the syntactic categories on the right-hand side of the rules. Consider the rule `vp(vp(V,NP)) -> v(V),np(NP)`. When we make a query using this DCG, the `V` in `v(V)` and the `NP` in `np(NP)` will be instantiated to terms representing parse trees. For example, perhaps `V` will be instantiated to

```
v(shoots)
```

and `NP` will be instantiated to

```
np(det(a),n(woman)).
```

What is the term corresponding to a vp made out of these two structures? Obviously it should be this:

```
vp(v(shoots),np(det(a),n(woman))).
```

And this is precisely what the extra argument `vp(V,NP)` in the rule `vp(vp(V,NP)) -> v(V),np(NP)` gives us: it forms a term whose functor is `vp`, and whose first and second arguments are the values of `V` and `NP` respectively. To put it informally: it plugs the `V` and the `NP` terms together under a `vp` functor.

To parse the sentence "A woman shoots" we pose the query:

```
?- s(T,[a,woman,shoots],[]).
```

That is, we ask for the extra argument `T` to be instantiated to a parse tree for the sentence. And we get:

```
T = s(np(det(a),n(woman)),vp(v(shoots)))
yes
```

Furthermore, we can generate all parse trees by making the following query:

```
?- s(T,S,[]).
```

The first three responses are:

```
T = s(np(det(the),n(woman)),vp(v(shoots),np(det(the),n(woman))))
S = [the,woman,shoots,the,woman] ;

T = s(np(det(the),n(woman)),vp(v(shoots),np(det(the),n(man))))
S = [the,woman,shoots,the,man] ;

T = s(np(det(the),n(woman)),vp(v(shoots),np(det(a),n(woman))))
S = [the,woman,shoots,a,woman]
```

This code should be studied closely: it's a classic example of building structure using unification.

Extra arguments can also be used to build **semantic representations**. We did not say anything about what the words in our little DCG mean. In fact, nowadays a lot is known about the semantics of natural languages, and it is surprisingly easy to build semantic representations which partially capture the meaning of sentences or entire discourses. Such representations are usually expressions of some formal language (for example first-order logic, discourse representation structures, or a database query language) and they are usually built up **compositionally**. That is, the meaning of each word is expressed in the formal language; this meaning is given as an extra argument in the DCG entries for the individual words. Then, for each rule in the grammar, an extra argument shows how to combine the meaning of the two subcomponents. For example, to the rule `s -> np, vp` we would add an extra argument stating how to combine the `np` meaning and the `vp` meaning to form the `s` meaning. Although somewhat more complex, the semantic construction process is quite like the way we built up the parse tree for the sentence from the parse tree of its subparts.

8.1.3 Beyond context free languages

In the previous lecture we introduced DCGs as a useful Prolog tool for representing and working with context free grammars. Now, this is certainly a good way of thinking about DCGs, but it's not the whole story. For the fact of the matter is: DCGs can deal with a lot more than just context free languages. The extra arguments we have been discussing (and indeed, the extra tests we shall introduce shortly) give us the tools for coping with any computable language whatsoever. We shall illustrate this by presenting a simple DCG for the formal language $a^n b^n c^n$.

The formal language $a^n b^n c^n$ consists of all non-null strings made up of `a`s, `b`s, and `c`s which consist of an unbroken block of `a`s, followed by an unbroken block of `b`s, followed by an unbroken block of `c`s, all three blocks having the same length. For example, `abc`, and `aabbcc` and `aaabbbccc` all belong to $a^n b^n c^n$. Furthermore, ε belongs to $a^n b^n c^n$.

The interesting thing about this language is that it is *not* context free. Try whatever you like, you will not succeed in writing a context free grammar that generates precisely these

strings. Proving this would take us too far afield, but the proof is not particularly difficult, and you can find it in many books on formal language theory.

On the other hand, as we shall now see, it is very easy to write a DCG that generates this language. Just as we did in the previous lecture, we shall represent strings as lists; for example, the string `abc` will be represented using the list `[a,b,c]`. Given this convention, here's the DCG we need:

```
s(Count) --> ablock(Count),bblock(Count),cblock(Count).

ablock(0) --> [].
ablock(succ(Count)) --> [a],ablock(Count).

bblock(0) --> [].
bblock(succ(Count)) --> [b],bblock(Count).

cblock(0) --> [].
cblock(succ(Count)) --> [c],cblock(Count).
```

The idea underlying this DCG is fairly simple: we use an extra argument to keep track of the length of the blocks. The `s` rule simply says that we want a block of `a`s followed by a block of `b`s followed by block of `c`s, and all three blocks are to have the same length, namely `Count`.

But what should the values of `Count` be? The obvious answer is: `1`, `2`, `3`, `4`,..., and so on. But as yet we don't know how to mix DCGs and arithmetic, so this isn't very helpful. Fortunately there's an easier (and more elegant) way. Represent the number 0 by `0`, the number 1 by `succ(0)`, the number 2 by `succ(succ(0))`, the number 3 by `succ(succ(succ(0)))`,..., and so on, just as we did it in Chapter 3. (You can read `succ` as "successor of".) Using this simple notation we can "count using matching".

This is precisely what the above DCG does, and it works very neatly. For example, suppose we pose the following query:

```
?- s(Count,L,[]).
```

which asks Prolog to generate the lists `L` of symbols that belong to this language, and to give the value of `Count` needed to produce each item. Then the first three responses are:

```
Count = 0
L = [] ;
```

```
Count = succ(0)
L = [a, b, c] ;

Count = succ(succ(0))
L = [a, a, b, b, c, c] ;

Count = succ(succ(succ(0)))
L = [a, a, a, b, b, b, c, c, c]
```

The value of `Count` clearly corresponds to the length of the blocks.

So: DCGs are not just a tool for working with context free grammars. They are strictly more powerful than that, and (as we've just seen) part of the extra power comes from the use of extra arguments.

8.2 Extra goals

Any DCG rule is really syntactic sugar for an ordinary Prolog rule. So it's not really too surprising that we're allowed to make use of extra arguments. Similarly, it shouldn't come as too much of a surprise that we can also add calls to any Prolog predicate whatsoever to the right hand side of a DCG rule.

The DCG of the previous section can, for example, be adapted to work with Prolog numbers instead of the successor representation of numbers by using calls to Prolog's built-in arithmetic functionality to add up how many `a`s, `b`s, and `c`s have already been generated. Here is the code:

```
s --> ablock(Count),bblock(Count),cblock(Count).

ablock(0) --> [].
ablock(NewCount) --> [a],ablock(Count), {NewCount is Count + 1}.

bblock(0) --> [].
bblock(NewCount) --> [b],bblock(Count), {NewCount is Count + 1}.

cblock(0) --> [].
cblock(NewCount) --> [c],cblock(Count), {NewCount is Count + 1}.
```

These extra goals can be written anywhere on the right side of a DCG rule, but must stand between curly brackets. When Prolog encounters such curly brackets while translating a DCG into its internal representation, it just takes the extra goals specified between the curly

brackets over into the translation. So, the second rule for the non-terminal `ablock` above would be translated as follows:

```
ablock(NewCount,A,B) :-
       'C'(A, a, C),
        ablock(Count, C, B),
        NewCount is Count + 1.
```

If you play around with this DCG, you will find that there are actually some problems with it. In contrast to the one that we saw in the last section, this new version only works correctly when used in the recognition mode. If you try to generate with it, it will at some point enter an infinite loop. We won't bother to fix this problem here.

This possibility of adding arbitrary Prolog goals to the right hand side of DCG rules, makes DCGs very very powerful (in fact, we can do anything that we can do in Prolog) and is not used much. There is, however, one interesting application for extra goals in computational linguistics; namely that with the help of extra goals, we can seperate the rules of a grammar from lexical information.

8.2.1 Separating rules and lexicon

By 'separating rules and lexicon' we mean that we want to eliminate all mentioning of individual words in our DCGs and instead record all the information about individual words separately in a lexicon. <!- To see what is meant by this, let's return to our basic grammar, namely:

```
np - - > det,n.

vp - - > v,np.
vp - - > v.

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

n - - > [woman].
n - - > [man].

v - - > [shoots].
```

We are going to separate the rules form the lexicon. That is, we are going to write a DCG that generates exactly the same language, but in which no rule mentions any individual word. All the information about individual words will be recorded separately. –>

Here is an example of a (very simple) lexicon. Lexical entries are encoded by using a predicate `lex/2` whose first argument is a word, and whose second argument is a syntactic category.

```
lex(the,det).
lex(a,det).
lex(woman,n).
lex(man,n).
lex(shoots,v).
```

And here is a simple grammar that could go with this lexicon. Note that it is very similar to our basic DCG of the previous chapter. In fact, both grammars generate exactly the same language. The only rules that have changed are those, that mentioned specific words, i.e. the `det`, `n`, and `v` rules.

```
det --> [Word],{lex(Word,det)}.
n --> [Word],{lex(Word,n)}.
v --> [Word],{lex(Word,v)}.
```

Consider the new `det` rule. This rule part says "a `det` can consist of a list containing a single element `Word`" (note that `Word` is a variable). Then the extra test adds the crucial stipulation: "so long as `Word` matches with something that is listed in the lexicon as a determiner". With our present lexicon, this means that `Word` must be matched either with the word "a" or "the". So this single rule replaces the two previous DCG rules for `det`.

This explains the "how" of separating rules from lexicon, but it doesn't explain the "why". Is it really so important? Is this new way of writing DCGs really that much better?

The answer is an unequivocal "yes"! It's *much* better, and for at least two reasons.

The first reason is theoretical. Arguably rules should not mention specific lexical items. The purpose of rules is to list *general* syntactic facts, such as the fact that sentence can be made up of a noun phrase followed by a verb phrase. The rules for `s`, `np`, and `vp` describe such general syntactic facts, but the old rules for `det`, `n`, and `v` don't. Instead, the old rules simply list particular facts: that "a" is a determiner, that "the" is a determiner, and so on. From theoretical perspective it is much neater to have a single rule that says "anything is a determiner (or a noun, or a verb,...) if it is listed as such in the lexicon". And this, of course, is precisely what our new DCG rules say.

The second reason is more practical. One of the key lessons computational linguists have learnt over the last twenty or so years is that the lexicon is by far the most interesting, important (and expensive!) repository of linguistic knowledge. Bluntly, if you want to get

to grips with natural language from a computational perspective, you need to know a lot of words, and you need to know a lot about them.

Now, our little lexicon, with its simple two-place `lex` entries, is a toy. But a real lexicon is (most emphatically!) not. A real lexicon is likely to be very large (it may contain hundreds of thousands, or even millions, of words) and moreover, the information associated with each word is likely to be very rich. Our `lex` entries give only the syntactical category of each word, but a real lexicon will give much more, such as information about its phonological, morphological, semantic, and pragmatic properties.

Because real lexicons are big and complex, from a software engineering perspective it is best to write simple grammars that have a simple, well-defined way, of pulling out the information they need from vast lexicons. That is, grammar should be thought of as separate entities which can access the information contained in lexicons. We can then use specialized mechanisms for efficiently storing the lexicon and retrieving data from it.

Our new DCG rules, though simple, illustrate the basic idea. The new rules really do just list general syntactic facts, and the extra tests act as an interface to our (admittedly simple) lexicon that lets the rules find exactly the information they need. Furthermore, we now take advantage of Prolog's first argument indexing which makes looking up a word in the lexicon more efficient. First argument indexing is a technique for making Prolog's knowledge base access more efficient. If in the query the first argument is instantiated it allows Prolog to ignore all clauses, where the first argument's functor and arity is different. This means that we can get all the possible categories of e.g. `man` immediately without having to even look at the lexicon entries for all the other hundreds or thousands of words that we might have in our lexicon.

8.3 Concluding remarks

We now have a fairly useful picture of what DCGs are and what they can do for us. To conclude, let's think about them from a somewhat higher level, from both a formal and a linguistic perspective.

First the formal remarks. For the most part, we have presented DCGs as a simple tool for encoding context free grammars (or context free grammars enriched with features such as *subject* and *object*). But DCGs go beyond this. We saw that it was possible to write a DCG that generated a non context free language. In fact, *any program whatsoever* can be written in DCG notation. That is, DCGs are full-fledged programming language in their own right (they are Turing-complete, to use the proper terminology). And although DCGs are usually associated with linguistic applications, they can be useful for other purposes.

So how good are DCGs from a linguistic perspective? Well, mixed. At one stage (in the early 1980s) they were pretty much state of the art. They made it possible to code

complex grammars in a clear way, and to explore the interplay of syntactic and semantic ideas. Certainly any history of parsing in computational linguistics would give DCGs an honorable mention.

Nonetheless, DCGs have drawbacks. For a start, their tendency to loop when the goal ordering is wrong (we saw an example in the last lecture when we added a rule for conjunctions) is annoying; we *don't* want to think about such issues when writing serious grammars. Furthermore, while the ability to add extra arguments is useful, if we need to use lots of them (and for big grammars we will) it is a rather clumsy mechanism.

It is important to notice, however, that these problems come up because of the way Prolog interprets DCG rules. They are not inherent to the DCG notation. Any of you who have done a course on parsing algorithms probably know that all top-down parsers loop on left-cursive grammars. So, it is not surprising that Prolog, which interprets DCGs in a top-down fashion, loops on the left-recursive grammar rule `s -> s conj s`. If we used a different strategy to interpret DCGs, a bottom-up strategy e.g., we would not run into the same problem. Similarly, if we didn't use Prolog's built-in interpretation of DCGs, we could use the extra arguments for a more sophisticated specification of feature structures, that would facilitate the use of large feature structures.

DCGs as we saw them in this chapter, a nice notation for context free grammars enhanced with some features that comes with a free parser/recognizer, are probably best viewed as a convenient tool for testing new grammatical ideas, or for implementing reasonably complex grammars for particular applications. DCGs are not perfect, but they are very useful. Even if you have never programmed before, simply using what you have learned so far you are ready to start experimenting with reasonably sophisticated grammar writing. With a conventional programming language (such as C++ or Java) it simply wouldn't be possible to reach this stage so soon. Things would be easier in functional languages (such as LISP, SML, or Haskell), but even so, it is doubtful whether beginners could do so much so early.

9

Exercises

9.1 Day 1

Question: *Which of the following are syntactically correct Prolog terms (what kind?) and which are not?*
a) `vINCENT`, *b)* `_Jules`, *c)* `Footmassage`, *d)* `'Jules'`, *e)* `big kahuna burger`, *f)* `loves(Vincent,mia)`, *g)* `'loves(Vincent,mia)'`, *h)* `Butch(boxer)`

Answer: a) `vINCENT` is an atom: it starts with a lower case letter.
b) `_Jules` is a variable: it starts with _.
c) `Footmassage` is a variable: it starts with a capital letter.
d) `'Jules'` is an atom: it is enclosed between '.
e) `big kahuna burger` is not a Prolog term: variables can never contain blanks and atoms cannot either unless they start and end with '. Complex terms are always of the *functor*(*Args*).
f) `loves(Vincent,mia)` is a complex term. The functor is `loves`, the arity 2.
g) `'loves(Vincent,mia)'` is an atom: it is enclosed between quotes.
h) `Butch(boxer)` is not a term. It starts with a capital letter and can therefore not be an atom or a complex term. It cannot be a variable either because variables are not supposed to contain brackets.

Question: *Represent the following in Prolog: "Zed is dead."*

Answer: `dead(zed).`

Zed is one particular person, so we use a constant (starting with z, not Z) to represented him. And we use the predicate `dead/1` to represent the property of being dead.

Question: *Represent the following in Prolog: "Mia loves everyone who is a good dancer."*

Answer: `love(mia,X) :- good_dancer(X).`

Variables in Prolog clauses are universally quantified: for all `X`, if `good_dancer(X)` is true, then `love(mia,X)` is true.

Question: *Represent the following in Prolog: "Marsellus kills everyone who gives Mia a footmassage."*

Answer: `kill(marsellus,X) :- give_footmassage(X,mia).`

For all `X`, if `X` gives Mia a footmassage, then Marsellus kills `X`.

Question: *Represent the following in Prolog: "Jules eats anything that is nutritious or tasty."*

Answer:

```
eat(jules,X) :- nutritious(X).
eat(jules,X) :- tasty(X).
```

To express a disjuntion, use several clauses: one to describe each possibility. If something is nutritious, Jules eats it. And if something is tasty, Jules also eats it.

Another possibility is to use `;`, which means *or*: `eat(jules,X) :- nutritious(X) ; tasty(X).` The use of `;` can easily make programs unreadable.

Question: *How many facts, rules, clauses are there in the following knowledge base? How many predicates does it define?*

```
woman(vincent).
woman(mia).
man(jules).
person(X) :- man(X); woman(X).
loves(X,Y) :- knows(Y,X).
father(Y,Z) :- man(Y), son(Z,Y).
father(Y,Z) :- man(Y), daughter(Z,Y).
```

Answer: There are 3 facts and 4 rules, which makes together 7 clauses. The knowledge base defines 5 predicates: `woman/1`, `man/1`, `person/1`, `loves/2`, `father/2`.

Question: *Suppose we are working with the following knowledge base:*

```
wizard(ron).
hasWand(harry).
quidditchPlayer(harry).
wizard(X) :- hasBroom(X),hasWand(X).
hasBroom(X) :- quidditchPlayer(X).
```

How does Prolog respond to the query `wizard(ron)`*?*

Answer: It answers `yes`. `wizard(ron)` is a fact in the knowledge base.

Question: *Given the same knowlege base, how does Prolog respond to the query* `wizard(harry)`*?*

Answer: It answers `yes`. The knowledge base contains the fact `quidditchPlayer(harry)`. It also tells us that quidditch players have brooms (5th clause). So, `hasBroom(harry)` is also true. Furthermore, the knowledge base contains the fact `hasWand(harry)`, and a rule that everybody who has a broom and a wand is a wizard (4th clause).

Question: *Given the same knowlege base, how does Prolog respond to the query* `wizard(hermione)`*?*

Answer: It answers `no`. The knowledge base does not contain anything that would allow us to infer that Hermione is a wizard.

9.2 Day 2

Question: *Which of the following pairs of terms match?*
a) `bread = bread`, *b)* `Bread = bread`, *c)* `'Bread' = bread`, *d)* `food(X) = food(bread)`,
e) `food(bread,X,beer) = food(Y,kahuna_burger)`,
f) `food(bread,X,beer) = food(Y,sausage,X)`,
g) `meal(food(bread),drink(beer)) = meal(X,Y)`, *h)* `food(X) = X`

Answer: a) `bread = bread` matches.
b) `Bread = bread` matches; the variable Bread gets instantiated with the atom bread.
c) `'Bread' = bread` doesn't match; `'Bread'` and `bread` are both atoms, but different ones.
d) `food(X) = food(bread)` matches; X gets instantiated with bread.
e) `food(bread,X,beer) = food(Y,kahuna_burger)` doesn't match; the complex terms don't have the same arity.
f) `food(bread,X,beer) = food(Y,sausage,X)` doesn't match; X cannot be instantiated with sausage as well as beer.
g) `meal(food(bread),drink(beer)) = meal(X,Y)` matches; X gets instantiated with food(bread) and Y with drink(beer).
h) `food(X) = X` matches; X is instantiated with the circular term food(food(food(...))). Note: This only matches because Prolog doesn't do the occurs. Otherwise it would notice that `food(X)` contains the variable `X` and can, therefore, not be unified with `X`.

Question: *Here is a tiny lexicon and mini grammar with only one rule which defines a sentence as consisting of five words: an article, a noun, a verb, and again an article and a noun.*

```
word(article,a).
word(article,every).
word(noun,criminal).
word(noun,'big kahuna burger').
word(verb,eats).
word(verb,likes).

sentence(Word1,Word2,Word3,Word4,Word5) :-
     word(article,Word1),
     word(noun,Word2),
     word(verb,Word3),
     word(article,Word4),
     word(noun,Word5).
```

What are the first four answers that Prolog gives when asked the query `sentence(A, B, C, D, E)`*?*

Answer:

```
A = a, B = criminal, C = eats, D = a, E = criminal ;
A = a, B = criminal, C = eats, D = a, E = 'big kahuna burger' ;
A = a, B = criminal, C = eats, D = every, E = criminal ;
A = a, B = criminal, C = eats, D = every, E = 'big kahuna burger' ;
```

For the first answer, Prolog instantiates `A` with the first article it finds in its knowledge base, `B` with the first noun it finds, `C` with the first verb, `D` with the first article again, and `E` with the first noun. When forced to backtrack, Prolog examines its last decision, i.e., to instantiate `E` with `criminal` and sees whether it can do something different. It can, namely to use `'big kahuna burger'` instead of criminal, and this is the second answer. For the third answer, Prolog again checks whether there are any other possibilities for its last decision, which was to instantiate `E`. There are none this time. So, it goes back to the second to last decision, which was to instantiate `D` with `a`. (Note that the binding of `E` gets undone.) There is an other possibility for `D`: `every`. So, Prolog instantiates `D` with `every`, and then chooses the first noun that it can find in its knowledge base for `E`. This is the third answer.

Question: *Given the following predicate definition, what will Prolog print onto the screen if asked the query*

```
?- greater_than(succ(succ(succ(succ(0)))), succ(succ(succ(0))))

greater_than(succ(X),0) :-
     write(succ(X)), write(' is greater than 0.').
greater_than(succ(X),succ(Y)) :-
     write('Is '), write(succ(X)),
     write(' greater than '), write(succ(Y)), write('?'), nl,
     greater_than(X,Y),
     nl,write(succ(X)), write(' is greater than '),
     write(succ(Y)),write('.').
```

`write/1` *is a built-in predicate for printing terms to the screen. For example,* `write('big kahuna burger')` *causes Prolog to write* `big kahuna burger` *and* `X = a, write(X)` *makes Prolog write* `a` *onto the screen.* `nl/0` *is a built-in predicate that writes a line break.*

Answer: Prolog prints the following onto the screen.

```
Is succ(succ(succ(succ(0)))) greater than succ(succ(succ(0)))?
Is succ(succ(succ(0))) greater than succ(succ(0))?
Is succ(succ(0)) greater than succ(0)?
succ(0) is greater than 0.
succ(succ(0)) is greater than succ(0).
succ(succ(succ(0))) is greater than succ(succ(0)).
succ(succ(succ(succ(0)))) is greater than succ(succ(succ(0))).
```

9.3 Day 3

Question: *How das Prolog respond to the following queries?*
a) `X is 3*4`, *b)* `X = 3*4`, *c)* `3 is X+2`, *d)* `1+2 is 1+2`, *e)* `*(7,+(3,2)) = 7*(3+2)`.

Answer: a) Prolog answers `X = 12.` It evaluates `3*4` and assignes the result to `X`.
b) Prolog answers `X = 3*4.` Variable `X` is instantiated with the complex term `3*4. 3*4` is *not* evaluated.
c) Prolog answers `ERROR: Arguments are not sufficiently instantiated.` All variables in the right argument of `is` must be instantiated by the time `is` is evaluated.
d) Prolog answers `no.` Prolog evaluates the arithmetic expression to the right of `is.` Then it tries to match the result to the term to the left of `is.` This fails as the number `3` does not match the complex term `1+2.`
e) Prolog answers `yes. 7*(3+2)` is just a user friendly way of writing `*(7,+(3,2))`.

Question: *Which of the following are syntactically correct representations of lists? How many elements do the syntactically correct lists have?*
a) `[ ]`, *b)* `[[1,2,3]|[ ]]`, *c)* `[[ ]|[ ]]`, *d)* `[1,2|[3,4]]`, *e)* `[1|2,3,4]`.

Answer: a) This is a correct representation of a list with zero elements. In fact, this is the only correct way of representing the empty list.
b) Correct — one element. The only element is again a list, namely the list `[1,2,3]`. Another way of representing the same list would be `[[1,2,3]]`.
c) Correct — one element, namely the empty list. Another way of representing the same list would be `[[ ]]`.
d) Correct — 4 elements. Another representation of the same list would be `[1,2,3,4]`.
e) *Not* correct. The tail of the list, i.e., what comes to the right of `|`, has to be a list again. `2,3,4` is not a list. However, `[2,3,4]` is a list, such that `[1|[2,3,4]]` would be a correct representation of a list with 4 elements.

Question: *How would Prolog respond to the following queries?*
a) `[a,b,c,d] = [a,[b,c,d]]`, *b)* `[a,b,c,d] = [a|[b,c,d]]`, *c)* `[ ] = [_|[ ]]`, *d)* `[a|[b|[c|[d|[ ]]]]] = [a,b,c,d]`, *e)* `X = [[a|[ ]]|[ ]]`

Answer: a) Prolog would answer `no` because the first list has four elements (`a, b, c`, and `d`) and the second one only two (`a` and the list `[b,c,d]`).
b) Prolog would answer `yes`.
c) Prolog would answer `no` because the first list has zero elements and the second list has one element (_).
d) Prolog would answer `yes` because `[a|[b|[c|[d|[ ]]]]] = [a|[b|[c|[d]]]] =`

`[a|[b|[c,d]]] = [a|[b,c,d]] = [a,b,c,d]`.
e) Prolog would answer `X = [[a]]` because `[[a|[ ]]|[ ]] = [[a]|[ ]] = [[a]]`.

Question: *Given the following definitions of the predicates* `a/1` *and* `b/1`*, which are the properties that lists need to have in order to satisfy these predicates? That is, given a list* `L`*, for what kinds of* `L` *is* `a(L)` *or* `b(L)` *true?*

```
a([ ]).
a([_| Rest]) :- b(Rest).
b([ _| Rest]) :- a(Rest).
```

Answer: `a(L)` is true if `L` is either the empty list, or if `L` has an even number of elements. `b(L)` is true if `L` has an odd number of elements.

Question: *Consider the following two predicate definitions.*

```
addone1([ ],[ ]).
addone1([HIn | TIn],[HOut | TOut]) :-
     HOut is HIn + 1,
     addone1(TIn,TOut).

addone2([ ],[ ]).
addone2([HIn | TIn],[HIn + 1 | TOut]) :-
     addone2(TIn,TOut).
```

How do the predicates `addone1` *and* `addone2` *differ? How will Prolog respond to the queries* `addone1([1,2,3,4],L)` *and* `addone2([1,2,3,4],L)`*?*

Answer:

```
?- addone1([1,2,3,4],L).
L = [2, 3, 4, 5]
?- addone2([1,2,3,4],L).
L = [1+1, 2+1, 3+1, 4+1]
```

`addone1` takes a list of numbers and adds 1 to each element of the list. The recursive clause of `addone1` says that the head of the output list (`HOut`) is the result of adding 1 to the head of the input list (`HOut is HIn + 1`). `addone2` takes a list of terms and builds a complex terms of the form `+(T,1)` from each element `T` of the input list. The recursive clause of `addone2` says that the head of the output list should be instantiated to the complex term `HIn + 1`.

9.4 Day 4

Question: *Given the following DCG, what does the query look like that you have to ask Prolog in order to find out a) whether "*the woman shoots*" is a sentence and b) whether "*the man*" is an NP? c) Which query makes Prolog list all VPs that that the grammar generates?*

```
s - -> np,vp.
np - -> det,n.
vp - -> v,np.
vp - -> v.
det - -> [the].
det - -> [a].
n - -> [woman].
n - -> [man].
v - -> [shoots].
```

Answer: To test whether "*the woman shoots*" is a sentence you ask `s([the,woman,shoots],[ ])`. Remember that the internal representation of `s - -> np,vp` is `s(A,B) :- np(A,X), vp(X,B)`, where `A` is the incoming list of words and `B` is what is left over of that list after Prolog has "bitten off" a sentence. You could also ask `s([the,woman,shoots,mumble,mumble],[mumble,mumble])` to find out whether "*the woman shoots*" is a sentence.

Accordingly, you ask `np([the,man],[ ])` to test whether "*the man* is an NP.

To make Prolog list all the VPs that the DCG can generate you ask `vp(X,[ ])`.

Question: *What do the strings look like that are generated by the following context-free grammar?*

$$
\begin{array}{lcl}
X & \rightarrow & Y\ a \\
X & \rightarrow & X\ b \\
X & \rightarrow & \varepsilon \\
Y & \rightarrow & Y\ b \\
Y & \rightarrow & X\ a
\end{array}
$$

Answer: The grammar generates strings that consist of *a*'s and *b*'s and contains an even number of *a*'s. The empty string is also accepted. So, *ababbbb* is accepted, but *bbabb* is not.

Question: *The direct translation of the context-free grammar given above into a DCG would look as follows.*

```
x - -> x, [b].
x - -> y, [a].
x - -> [ ].
y - -> y, [b].
y - -> x, [a].
```

What problems does this DCG have?

Answer: The grammar is *left-recursive* (`x - -> x, [b]` and `y - -> y, [b]`), so that Prolog might end up in an endless loop.

In this particular case, we can repair the DCG by simply swapping around the order of the goals on the right hand side of the arrow. This DCG recognizes the same language as the context-free grammar given above. However, the parse trees that it gives rise to look different.

```
x - -> [b], x.
x - -> [a], y.
x - -> [ ].
y - -> [b], y.
y - -> [a], x.
```

Another advantage of this solution is that Prolog doesn't have to guess whether the input string ends in `a` or `b` in order to decide which of the possible rules to use.

If we also want to use the grammar for generation, we also have to change the order to the rules, such that the recursive rules come last.

```
x - -> [ ].
x - -> [a], y.
x - -> [b], x.
y - -> [a], x.
y - -> [b], y.
```

9.5 Day 5

Question: *The following is a small DCG that uses an extra argument to construct the parse tree. What does the query look like that you have to ask Prolog in order to find out whether* "the woman shoots" *is a sentence?*

```
s(s(NPTree,VPTree)) - -> np(NPTree), vp(VPTree).
np(np(DETTree,NTree)) - -> det(DETTree), n(NTree).
vp(vp(VTree,NPTree)) - -> v(VTree), np(NPTree).
vp(vp(VTree)) - -> v(VTree).
det(det(the)) - -> [the].
det(det(a)) - -> [a].
n(n(woman)) - -> [woman].
n(n(man)) - -> [man].
v(v(shoots)) - -> [shoots].
```

Answer: You have to ask `s(Tree,[the,woman,shoots],[ ])`. The internal representation of `s(s(NPTree,VPTree)) - -> np(NPTree), vp(VPTree)` is

```
s(s(NPTree,VPTree), A, B) :-
     np(NPTree, A, X),
     vp(VPTree, X, B).
```

Question: *Given the DCG from the first question, how would Prolog respond to the following queries?*
a) s(Tree,[the,woman,shoots,the,man],[ ]).
b) s(Tree, Words, [ ]).
c) s(s(np(det(W1),n(W2)), vp(v(W3))), L, [ ]).

Answer: a) `Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(the), n(man))))`

b) `Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(the), n(woman))))`
`Words = [the, woman, shoots, the, woman] ;`
`Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(the), n(man))))`
`Words = [the, woman, shoots, the, man] ;`
`Tree = s(np(det(the), n(woman)), vp(v(shoots), np(det(a), n(woman))))`
`Words = [the, woman, shoots, a, woman] ;`
usw.

c) This query give us only four answers:

```
L = [the, woman, shoots] ;
L = [the, man, shoots] ;
L = [a, woman, shoots] ;
L = [a, man, shoots]
```

Question: *The following is an attempt to extend the DCG given above such that it can properly handle the pronouns* "he", "she", "him", "her"*? What mistakes can you find?*

```
s(s(NPTree,VPTree)) - -> np(NPTree, subj), vp(VPTree).
np(np(DETTree,NTree)) - -> det(DETTree), n(NTree).
np(np(PRONTree), CASE) - -> pron(PRONTree, CASE).
vp(vp(VTree,NPTree)) - -> v(VTree), np(NPTree, obj).
vp(vp(VTree)) - -> v(VTree).
det(det(the)) - -> [the].
det(det(a)) - -> [a].
n(n(woman)) - -> [woman].
n(n(man)) - -> [man].
v(v(shoots)) - -> [shoots].
pron(subj,pron(he)) - -> [he].
pron(subj,pron(she)) - -> [she].
pron(obj,pron(him)) - -> [him].
pron(obj,pron(her)) - -> [her].
```

Answer: 1) If we decide to introduce a feature for some category, we have to specify that feature everywhere where this category is used. So, even though the case feature is not relevant for NPs consisting of a determiner and a noun, we have to specify the case feature on the left hand side of the second rule. Since the value doesn't matter, we just use `_`.

2) The order in which we use the extra arguments that go with a particular category, always has to be the same. In the rule of the DCG given above, `pron` is used with two extra arguments: the first one is for assembling the syntactic structure and the second one for the case information. In the last rules of the DCG, these two features are interchanged.

Here are the corrections that have to be made.

```
...
np(np(DETTree,NTree), _) - -> det(DETTree), n(NTree).
...
pron(pron(he), subj) - -> [he].
pron(pron(she), subj ) - -> [she].
pron(pron(him), obj) - -> [him].
pron(pron(her), obj) - -> [her].
```
