This one’s about grammar and parsing. I’m not asking you to write any Prolog code this time: the task this time is to add lexical entries and rules to three grammars, and to test them out using a set of Prolog programs. What you have to add will be quite small, but the grammars are quite large, and you will have to read them and understand what they say, because you won’t be able to add to them unless you understand them. The intention is that making you think about these grammars will give you some insights into the nature of natural language grammars. There is stuff to read in the specifications of the three grammars themselves: reading and assimilating that stuff is what this lab is all about. There’s a lot to read here, not all that much to do.

To load the grammars and the parser, you consult a top-level file called setup.pl. Once you’ve done that, doing

```
| ?- setup(lab3a).
```

will read the grammar for exercise (a) (lab3b, lab3c for the others), and will then recompile the parser. It has to be done this way, because the parser pays attention to the sets of features used by the grammar, so you have to recompile it after you load the grammar itself. If you finish one exercise and want to move on to the next, and you do

```
| ?- setup(lab3b).
```

without exiting from Prolog first, you’ll get messages about various things being redefined. That’s fine—we’ve changed grammars, our rules and so on are defined in a different file, so we’d expect to get warning messages.

You can then parse a piece of text by doing

```
| ?- chart([the, cat, sat], P), pretty(P).
```

```
[[the, cat], sat]
P = [[the,cat],sat] ?
yes
```

(there’s not much point in the call of pretty here, but when they get more complicated it’s helpful) and you can get it to print out the edges that get made along the way by doing

```
| ?- set(showSteps).
```

(you only have to do this once. If you want to stop edges being printed do unset(showEdges)).

The grammars all come in four sections:
Signature

A grammar file starts with a signature to say what features are being used in this grammar. This looks like

\[
\text{signature(sign=[\text{cat=}_, \text{agree=}_, \text{case=}_, \text{subcat=}_, \text{moved}=[\text{before=}_, \text{after=}_])].}
\]

This one says that a sign has the features \text{cat}, \text{agree}, \text{case}, \text{subcat} and \text{moved}, and that \text{moved} has two subfeatures called \text{before} and \text{after}. You’ve seen all of these apart from \text{moved} in the lectures, and you can ignore \text{moved} : it’s in there because the parser is written to allow for ‘movement’, because we’re going to need that later, but for now you can just ignore it.

You can access a feature that has been specified in the current signature by writing, for instance,

\[
| ?- \text{cat@X -- n}.\]

This notation is a ‘Prolog extension’: one of the advantages of languages which allow programs to be treated as data structures is that you can easily extend them, because you can transform the data structures that make up a program before they are compiled. The programs used for these labs make quite a lot of use of this idea, with the \text{feature@SIGN} notation being an important example.

There is another useful extension which lets us say that several items have the same value for some feature, though we may not know what that value is:

\[
| ?- \text{agree :: [X, Y, Z]}.\]

says that \text{X}, \text{Y} and \text{Z} have the same value for \text{agree}.

Types

We then get a set of ‘type definitions’ which give names to particular configurations of features. It is generally better to use these type definitions in rules and descriptions of dictionary entries than to specify values for features directly, since it makes it easier to change things later. It also makes things a bit more readable, which is nice, but the real advantage is that we can change the definition of a type once, and have that change reflected everywhere the type is used, rather than having to go and change all the rules and lexical entries that make use of it.

The type definitions are written as ordinary Prolog rules, but you can apply them using another Prolog extension. The rules below, for instance, define two simple types:
np(X) :-
    cat@X -- np.

subjCase(X) :-
    case@X -- subj.

I can use them to describe a subject-case NP by writing

subjnp(X) :-
    X <> [np, subjCase].

This rule should be read as saying that X is a subjnp if it is in the types np and subjCase. You can put as many types as you want in the list on the right-hand side of <>. If you only want one you can omit the list brackets.

Rules

Rules are as we’ve seen them in the lectures. The rule

S ==> [NP, VP] :-
    S <> s,
    NP <> [np, subjCase],
    VP <> vp,
    agree :: [NP, VP].

for instance says that you can make a sentence out of a subject-case NP and a VP if the NP and VP share the same value for agree. It is important to remember that the S, NP and VP in the rule S ==> [NP, VP] are variables, and that the type specifications S <> s and so on say what the items denoted by these variables are like. If I just wrote

S ==> [NP, VP].

I’d be saying that anything can be made out of anything followed by anything else, since I wouldn’t have said what properties these items had. NP is a variable. The Prolog interpreter does not know that I’ve used a name for this variable that indicates that I want it to denote an NP. You have to specify the required properties explicitly in the rule.
Lexicon

And then finally we get the dictionary. I’m calling the things in the dictionary ‘lexical entries’ rather than ‘words’ because for exercise (b) I’m going to want to put things in it that aren’t really words, so I’m going to use a more general term from now on. But pretty much they’re just specifications of what I know about some word (or word-part, for exercise (b)).

```prolog
lexEntry(i, X) :-
    X <> [np, subjCase],
    agree@X -- firstsing.
```

Exercises

(a) Extending the basic feature-based grammar [6 marks]

The first exercise uses a close variation on the standard feature-based grammar from the lectures. You’ll probably find something very like the solution buried in the lecture handouts. This one is really just to get you used to what the grammars look like, and what’s involved in adding things to them.

The aim of this one is to add whatever is needed to enable the system to analyse the following sentences (remember that things beginning with upper case characters look like variables to Prolog, so it makes sense to have i rather than I):

1. I know she loves me.

   ```prolog
   | ?- chart([i, know, she, loves, me], P).
   P = [i,[know,[she,[loves,me]]]] ?
   yes
   ```

2. I gave her a present.

   ```prolog
   | ?- chart([i, gave, her, a, present], P).
   P = [i,[[gave,her],[a,present]]] ?
   yes
   ```

3. I gave it to her.

   ```prolog
   | ?- chart([i, gave, it, to, her], P).
   P = [i,[[gave,it],[to,her]]] ?
   yes
   ```
John loves Mary and Mary loves John.

\[
\text{\texttt{| ?- chart([john, loves, mary, and, mary, loves, john], P).}}
\]
\[
\text{\texttt{P = [[[john,[loves,mary]],and],[mary,[loves,john]]] ? yes}}
\]

he wore a red and black shirt.

\[
\text{\texttt{| ?- chart([he, wore, a, red, and, black, shirt], P).}}
\]
\[
\text{\texttt{P = [he,[wore,[a,[[[red,and],black],shirt]]]] ? yes}}
\]

Fix the grammar so it does **not** accept ‘her loves me’ or ‘I love she’.

For the first three of these, you need to extend the set of verb types (\texttt{intrans}, \texttt{trans}) so that they will allow for verbs that take a sentence, or two NPs, or an NP and a PP, as arguments. So you should look at the rules that mention these features, and at the lexical entries for \texttt{sleep} and \texttt{saw}, and try to adapt those to cover the examples.

The key to the other two is that we want ‘and’ to link any pair of items of the same kind to make a compound of the same type – two sentences (as in (4)), two adjectives (5), two verbs (‘He buys and sells stolen cars’), two NPs (‘I saw John and Mary’). You should be able to do it with one rule, and one entry for ‘and’, but it takes a bit of thought. The standard name for words like ‘and’ (think of some others) is ‘conjunction’, so you might want to set the \texttt{cat} for words like this to be \texttt{conj}.

(b) **Morphology, X-bar theory [7 marks]**

The second exercise covers two topics:

i Words are made out of smaller things. Can we deal with that in the same way that sentences are made out of smaller things?

ii NPs are based on nouns, sentences are based on VPs which are based on verbs, adjectival phrases are based on adjectives, … Can we capture this notion by splitting the \texttt{cat} feature into two pieces – the \texttt{cat} itself and a \texttt{level}?
The signature in \texttt{lab3b.pl} is 

\begin{verbatim}
signature(sign=[head=[cat=_, agree=[number=_, person=_], case=_], level=_], subcat=_], moved=[before=_, after=_])
\end{verbatim}

where the \texttt{cat} is grouped with a number of other features that seem to be specified by the head word of the phrase but are inherited by the bigger phrase itself (so \texttt{sees} is present tense and third person singular, so \texttt{sees the old man in the park} is present tense and third person singular), and there is a separate feature called \texttt{level}.

\texttt{level} has several possible values - \texttt{root}, \texttt{affix}, \texttt{word}, 1 and 2. The first three should be reasonably self-explanatory – \texttt{see} is a root, \texttt{ing} is an affix, \texttt{seeing} is a word. The other two apply to phrases, roughly saying whether they are the largest kind of phrase that you can base on this word (\texttt{level}=2 or not (\texttt{level}=1). So \texttt{John loves Mary} has \texttt{cat=v}, because it’s based on something verb-like (\texttt{loves}) and \texttt{bar}=2 (because you can’t add anything to it), whereas \texttt{loves Mary} also has \texttt{cat=v}, because it’s based on something verb-like (\texttt{loves}) but has \texttt{bar}=1 (because you haven’t got the subject yet).

The grammar in \texttt{lab3b.pl} has entries for word parts, and rules for combining them, in addition to rules for combing words and phrases. So if you do

\begin{verbatim}
| ?- chart([the, cat, s, walk, ed]).
\end{verbatim}

the system will analyse it as

\begin{verbatim}
[[the, [cat, s]], [[walk, ed]]]
\end{verbatim}

(it won’t deal with \texttt{[the, cats, walked]}, because it’s not doing ‘segmentation’, so don’t try that one).

This exercise has two parts:

- add rules to cover comparative and superlative adjectives, so we can do

\begin{verbatim}
(6) I saw the old man.
| ?- chart([i, saw, the, old, man], P).
P = [[i, [saw, [the, [old, man]]]] ? yes
\end{verbatim}
(7) I saw the older man.

| ?- chart([i, saw, the, old, er, man], P).
P = [i, [saw, [the, [[old, er], man]]]] ?
yes

(8) I saw the oldest man.

| ?- chart([i, saw, the, old, est, man], P).
P = [i, [saw, [the, [[old, est], man]]]] ?
yes

- add rules which treat PPs as the level=2 version of prepositions, in much the same way that we're treating sentences as the level=2 version of verbs and NPs as the level=2 version of nouns. How many analyses does your grammar assign to (10)? Some of these convey exactly the same information: pick two such analyses.

(9) I saw the man in the park.

| ?- chart([i, saw, the, man, in, the, park], P).
P = [i, [saw, [the, [man, [in, [the, park]]]]]] ? ;
P = [i, [[saw, [the, man]], [in, [the, park]]]] ? ;
no

(I’ve used ; to force multiple answers to this one, which is why it ends up with no – there aren’t any more solutions after the second one)

(10) I saw the rich old man with a big nose with a spot on it in the park with a telescope.
(using pretty(P), fail tells it to print out the answer in a readable format and then try to find another one)

| ?- chart([i, saw, the, rich, old, man, with, a, big, nose, with, a, spot, on, it, in, the, park], P), pretty(P), fail.
Virtually all the work in this exercise is about understanding what’s going on. Hint: you do not need any new rules to cover the comparative and superlative forms for adjectives, just new entries in the lexicon. You will need a rule for base-form adjectives, just like we have one for base-form nouns, and you will need a rule about making PPs.

(c) Categorial grammar [7 marks]

Categorial grammar has just two rules:

\[ \text{X0} \rightarrow [\text{X1}, \text{Y}] :\]
- \text{head@X0} -- head@X1,
- \text{args@X1} -- [Y | args@X0],
- \text{dir@Y} -- after.
X0 ==> [Y, X1] :-
    head@X0 -- head@X1,
    args@X1 -- [Y | args@X0],
    dir@Y -- before.

The first says that if you’ve got an item of type X1 which says that it wants a following Y, and there is a following Y, then you can combine them to form an item of the same basic kind but which no longer needs a Y. The second says much the same for cases where X1 wanted a preceding Y.

When you use this kind of grammar, the definitions of individual words become much more complex. After all, most of the grammar has disappeared, or at least been hidden. The place where it’s been hidden is in the definitions of the individual words.

If you look at the lexical entries for this grammar, they look just like the ones in lab3a.pl, saying that ‘saw’, for instance, is a transitive verb. So where has the complexity gone? Into the definitions of things like tverb, where the list args and the direction in which they are expected to appear is specified.

Add definitions of scompverb, adj and conj so that this grammar can cope with (11)–(15). You do not need to add new rewrite rules. Everything in categorial grammar is handled by getting the list of args right.

(11) I know she loves me.

| ?- chart([i, know, she, loves, me], P).
| P = [i, [know, [she, [loves, me]]]] ? ;
| no

(12) I saw the old man.

| ?- chart([i, saw, the, old, man], P).
| P = [i, [saw, [the, [old, man]]]] ? ;
| no

(13) I saw the rich old man.

| ?- chart([i, saw, the, rich, old, man], P).
| P = [i, [saw, [the, [rich, [old, man]]]]] ? ;
| no

(14) John loves Mary and Mary loves John.
\[ ?- \text{chart([john, loves, mary, and, mary, loves, john], P).} \]
\[ P = [[\text{john}, [\text{loves, mary}]], [\text{and}, [\text{mary}, [\text{loves, john}]]]] \] ? ;
\[ \text{no} \]

(15) I wore a red and black shirt.

\[ ?- \text{chart([he, wore, a, red, and, black, shirt], P).} \]
\[ P = [\text{he, [wore, [a, [[\text{red, [and, black]}], \text{shirt}]]]}] \] ? ;
\[ \text{no} \]

The conjunction here should be of red and black. This is quite tricky. It’s an interesting case, but don’t kill yourself trying to get it right – it’s only worth 1 mark!