Manual for syntactic annotators

Table of Contents

Authors 1
Introduction 1
- Starting Alpino 1
- File format of suites 2
- File format of annotations 3
- A simple example of the annotation process 4
- Options 4
- Tips and tricks 5

POS-tagging 7
- Verbs 8
- Particles 9
- Fixed 9
- Nouns and related tags 9
- Verbal nouns 10
- Determiner 11
- Pronouns 11
- Adjectives 12
- Prepositions 13
- Adverbs 13
- Complementizers 14
- Conjunctions 14
- UNKNOWN words 14
- Proper names 14

Authors

Bart Cramer <B.Cramer@student.rug.nl>

Gertjan van Noord <vannoord@let.rug.nl>

Introduction

This document gives an overview of the Alpino system and related software. Together, these programs can be used to annotate a corpus of Dutch sentences syntactically. We will not discuss the specific rules of syntactic annotation itself; for this we refer to the CGN document. Alpino itself is a parser which uses linguistic knowledge and various heuristics to construct appropriate linguistic structures of Dutch sentences.

Starting Alpino

Alpino is the tool that we use for interactive annotation. Before using Alpino, make sure that Alpino is properly installed on your machine. Please consult a local wizard for this. The Alpino program itself can be started with the command:

```
Alpino
```

The command can take very many options. However, for annotation, there is a specialized script, called Annotate, that we normally use for the specific purpose of annotating. It requires an argument which indicates which suite (a named collection of sentences) you want to annotate.

```
Annotate NameOfSuite
```

Annotate can take all the options that Alpino knows about, but these options are placed *after* the name of the suite.

The Annotate script has some particular assumptions about the directory structure that you are working in. You are advised to construct a new directory for the purpose of annotation. Perhaps this directory is called `DCOI` or `LASSY` or `ALPINO` or some such. In this directory, there ought to be two sub-directories called `Suites` and `Treebank` respectively. In the `Suites`-directory you will collect the files (called `suites`) that you need to annotate. The annotations will be placed in the directory `Treebank`. The `Annotate` script will create a sub-directory in `Treebank` for each suite that you annotate.

File format of suites

In the directory `Suites` you need to place the sets of sentences that are to be annotated. Each such a set is called a suite. Such a set of sentences is placed in a file ending with the extension `.sents`. Although one is free to use any name for the suites, there should be no dots in the filenames (e.g. `1.1.1.2.Thesis.txt.tok.pl`), as these will confuse Alpino. For instance, you might create the file my_corpus.sents containing the following lines:

```
1|De zon komt op boven de stortplaats van het stadje Merlijn .
2|Er klinkt een zacht geluid .
3|Lila het vosje spitst de oren .
4|Ze probeert erachter te komen waar het geluid vandaan komt .
5|Nee , het is geen vogel .
```

Every line contains a key, a vertical bar, and the tokens of the sentence, separated by space. Note that the sentences are assumed to be tokenized here already: punctuation symbols are treated as separate tokens and therefore seperated by spaces as well. There are various tools in the Alpino software suite for tokenizing texts into this format, but here we assume that you have been given files such as those already.

The keys (whatever is placed to the left of the vertical bar | on each line) can be arbitrary strings, and are used to identify each sentence. Later, the file name containing the annotation of each sentence will use this key.

If you have the correct `Makefile` in the Suites directory, you can give compile the corpus into a format readable by Alpino. We use the `make' program to create the compiled corpus. The `make' program requires a file called `Makefile` which should contain the following definition:

```
%.pl : %.sents
        echo ":- module(suite,[ sentence/2 ])." > $*.pl
        echo >> $*.pl
        cat $*.sents | grep .|\
        sed -e "s/| /|/" \
            -e "s/  *$$//" \
            -e "s/\\\\/\\\\\\\\/g" \
            -e "s/'/\\\\'/g" \
            -e "s/ /','/g" |\
        awk -F\| ' !/^(%|--)/ {\
        printf("sentence(%c%s%c,[%c%s%c]).\n",39,$$1,39,39,$$2,39); N++;}' |\
            sicstus -l $(ALPINO_HOME)/Suites/echo >>$*.pl
```

Don't worry about the details of this unreadable definition. It is meant to treat white space, quotes etc, in a way so as not to confuse Prolog (Alpino is implemented in Prolog). You can copy a file with this definition from the Alpino distribution:

```
cp $ALPINO_HOME/Makefile.suite Makefile
```

You can now use this Makefile in order to create the Prolog version of the suite:

```
make my_corpus.pl
```

If all goes well, you now have the file my_corpus.pl which looks as follows:

```
:- module(suite,[ sentence/2 ]).

sentence('1',['De',zon,komt,op,boven,de,stortplaats,van,het,stadje,'Merlijn','.']).
sentence('2',['Er',klinkt,een,zacht,geluid,'.']).
sentence('3',['Lila',het,vosje,spitst,de,oren,'.']).
sentence('4',['Ze',probeert,erachter,te,komen,waar,het,geluid,vandaan,komt,'.']).
sentence('5',['Nee',',',het,is,geen,vogel,'.']).
```

If you need to change the `suites` file, then you should repeat the `make` compilation step, in order that the changes are also propagated to the compiled version of the suite.

File format of annotations

The annotations are placed in the Treebank subdirectory. Because each sentence gets its own file, a new subdirectory is needed for each suite. In there, the annotation of a sentence is saved as an xml-file, example `Treebank/my_corpus/2.xml` (Of course, if you haven't done any annotations yet, then there will not be any .xml files in your directory). This file could look as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<alpino_ds version="1.0">
  <node id="0" rel="top" cat="top" begin="0" end="6">
    <node id="1" rel="--" cat="smain" begin="0" end="5">
      <node id="2" rel="mod" pos="adv" begin="0" end="1" root="er" word="Er"/>
      <node id="3" rel="hd" pos="verb" begin="1" end="2" root="klink" word="klinkt"/>
      <node id="4" rel="su" cat="np" begin="2" end="5">
        <node id="5" rel="det" pos="det" begin="2" end="3" root="een" word="een"/>
        <node id="6" rel="mod" pos="adj" begin="3" end="4" root="zacht" word="zacht"/>
        <node id="7" rel="hd" pos="noun" begin="4" end="5" root="geluid" word="geluid"/>
      </node>
    </node>
    <node id="8" rel="--" pos="punct" begin="5" end="6" root="." word="."/>
  </node>
  <sentence>Er klinkt een zacht geluid .</sentence>
</alpino_ds>
```

There are various programs closely related to treebanks: `dtview`, `dtedit` (also known as Thistle), `dtchecks`, `dtsearch` and `dttred`.

You can use dtview to display annotated sentences graphically. The `dtview` program has a rather intuitive interface, but there are a few short keyboard command that makes going through treebank a bit less damaging for your wrist: Page Up and Page Down for going to the next and previous tree, and the arrow keys for navigating in trees that don't fit on the screen. The *t* key triggers the same response as the *TrEd* button on the top, namely opening `dttred` for that particular tree. Here, you can edit the tree.

`dtchecks` is used for a superficial check on a treebank. Its syntax is `dtchecks directory`, to be run from the Treebank directory, and it is able to recognize errors like a node with only one child and incompatible sisters/children/parents. Annotators produce these flaws rather often, so before rounding off a corpus, you must use it. However, note that the dtchecks program might also complain about valid annotations.

`dtsearch` is a program that permits the user to search for trees in the treebank with certain criteria. By standard, it does it recursively in all subdirectories. After having retrieved the correct XML files, `dtsearch` will show them in dtview (if you use the -v option). For example, if you would like to find all trees in which *zoals* occurs as a preposition, you can issue the command:

```
dtsearch -v '//node[@word="zoals" and @pos="prep"]' .
```

The dtsearch program has a very powerful query language (known as XPATH), that is documented elsewhere.

A simple example of the annotation process

To see the system at work, let's try to see the procedure from the start until the end. First, make a Suite and a Treebank directory, and create or copy an example suite in your Suite directory, and compile that suite into Prolog format. Let's assume the suite is called `my_corpus`. Then, issue the command

```
Annotate my_corpus
```

from your current directory (that should now contain the Suites and Treebank sub-directories).

Alpino starts up its graphical environment. On the top you will see a series of menu buttons, which we will ignore for now (although it is perhaps useful to know that you can stop the system by selecting the File menu, and clicking the Halt item in that menu).

Below the menu buttons you see the `Parse` button on the left, with an empty long space to its right, followed by a shorter space at the right hand side. Once we have selected a sentence, the sentence will appear in this space, with its key displayed further to the right. You can try parsing a sentence by selecting it from the list box below the `Parse`-button, followed by pressing that `Parse`-button.

Before Alpino really starts parsing the sentence, it presents a new widget with the lexical analysis of the sentence. At this point, you can influence the lexical analysis phase of Alpino, but for now we simply click the Parse button (left below) on this new widget, to really get going. Below, the lexical analysis phase is explained in more detail (you can also swith off the interactive lexical assignment using the option `interactive_lexical_analysis=off`).

Alpino will now (finally!) parse the sentence syntactically, and it will show you the most probable analysis in the left pane. For each analysis, a green button with the number of the analysis is constructed as well. As a short-cut, clicking on such a numbered analysis button with your right mouse button will immediately display the corresponding dependency structure of that analysis. A left-click on the number will give you a menu, with which you can, for example, save your parse. If you select *XML -> Save*, Alpino will automatically create the needed file in the corresponding subdirectory of the Treebank folder. If the correct annotation is not available, you can select *XML -> Save and Edit*, to start the editor for this annotation. There are many more options, some of which are useful. Please experiment.

Options

In Alpino, there are a lot of options available. Some options have a quality/computational time trade-off.

The following are the most important:

- number_analyses. This sets the maximum number of analyses the parser will produce. The `-fast` option at start-up sets this option to 1. Unless dealing with very complex sentences, it is recommended to set this option to some small integer. If you increase this number, you should also increase the value of the `disambiguation_beam' flag to the same number. The special value *0* indicates that *all* analyses are produced.
- interactive_lexical_analysis. If turned on, it gives the user the possibility to adjust the suggested POS-tagging. When manually annotating, it is highly recommended to keep this on.
- fast. Using this option has a number of consequences. First of all, the POS-tagger is switched on the reduce lexical ambiguities. Furthermore, an efficient search algorithm is used to select the best parse. In this case, only the best parse is produced.
- slow. This is the alternative to `fast` (you guessed!). The POS-tagger is switched off, and all possible analyses are produced. This is only useful for shorter sentences.

Tips and tricks

Next to the general way of parsing sentences, there are a number of manual tricks available, which can help a great deal in improving the quality of the parses or in speeding up the parsing process.

- A large part of the computational time is spent on finding the correct constituents. The annotator can give hints to Alpino about this by putting straight brackets around constituents. Both brackets should be surrounded by a single space on both sides otherwise the POS-tagging will give problems, regarding a space as a word. This is a nice feature especially for attaching modifiers at the correct location, enumerations and complex nestings in syntactic structures. Brackets are normally associated with a category name, using the @-operator. Normal syntactic symbols can be used here as @np, @pp etc. Examples:

```
Hij fietst [ @pp op zijn gemakje ] [ door de straten van De Baarsjes ] .
FNB zet teksten van [ [ kranten en tijdschriften ] , boeken , [ studie- en vaklectuur ] , bla
[ @np De conferentie die betrekking heeft op ondersteunende technologie voor gehandicapten in
```

- You can also force a lexical assingment to a token or a series of tokens, if the Alpino lexicon does not contain the proper assignment. A @postag followed by an Alpino lexical category will force the assignment of the corresonding lexical category to the words contained in the brackets. This comes in handy when a part of the sentence is written in a foreign language or a spelling mistake has occurred. Of course, it may need adjusting with the editor afterwards.

```
Hij heeft een beetje [ @postag adjective(no_e(adv)) curly ] haar .
Op een [ @postag adjective(e) mgooie ] dag gingen ze fietsen .
Mijn [ @postag noun(de,count,sg) body mass index ] laat te wensen over .
```

- Also, if the annotator can predict that a certain token or series of tokens will make the annotation a mess, it can be skipped by @skip. In such a case, the sentence is parsed as if the word(s) that are marked with `skip` where not there, except that the numbering of the words in the resulting dependency structure is still correct. Clearly, in many cases an additional editing phase is required to obtain the fully correct analysis, but this method might reduce efforts considerably.

```
Ik wil [ @skip ??? ] naar huis
Ten opzichte [ @skip echter ] van deze bezwaren willen wij ....
```

- A related trick that is useful in particular cases, is to add a word to the sentence in order to ensure that the parse can succeed, but instruct Alpino that this word should not be part of the resulting dependency structures. Such words are labeled phantom as follows:

```
Ik aanbad [ @phantom hem ] dagelijks in de kerk
Ik kocht boeken en Piet [ @phantom kocht ] platen
Ik heb [ @phantom meer ] boeken gezien dan hem
```

Limitations: a phantom bracketed string can only contain a single word. The technique does not work yet for words that are part of a multi-word unit.

Warning: the resulting dependency structure is most likely not well-formed and often needs manual editing.

- After annotating a sentence, it might turn out that you used insufficient or incorrect brackets to get to the right annotation. In this case, you may not want to re-do the interactive lexical assignment. Middle-clicking *Parse* will parse the sentence, using the lexical assigment of the previous try.

- In the right bottom of the Alpino screen, there are two extra buttons. *Annotate shortest* is particularly useful for beginning annotators: this button will start to annotate the shortest yet unparsed sentence in the corpus. *Annotate next* does what the name suggests it does: it annotates the first sentence in the corpus that is still unparsed. This is a nice feature to check whether you didn't forget a sentence in the corpus. As a short-cut, clicking the now familiar `Parse` button with your rightmost mouse-button has a similar effect by selecting the next sentence to be annotated.

- Sometimes you would like to check if a sentence is already done, or you would like to look back at previous sentences to ensure coherence between sentences. For this, you can use the *Treebank* button; this will activate the `dtview` program on the analysis of the current sentence.

- There is a small bug in Alpino: when it is being used for a long time, the canvas for drawing the trees is full, and these drawings will render the newly generated trees unreadable. You can clear the canvas by clicking on it with the rightmost mouse button.

- Although most work is done in the graphical interface, some things can only be done in the Prolog interpreter screen (the terminal). One of the most used options is to quickly check the possible POS-tags of a word form. This can be done with the command *mlex word*. Quickly retrieving the parse of one sentence can be done as well, using the * command. Quitting from the terminal mode is done with the command *quit* (without a period, as is expected in a Prolog environment). Also, `sen 1130` retrieves sentence 1130 from the corpus.

```
1 |: mlex dreun
stem           string         his                 cat
dreun          [dreun]        normal              noun(de,count,sg)
dreun          [dreun]        normal              verb(hebben,sg1,intransitive)
dreun          [dreun]        normal              verb(hebben,sg1,sbar)
```

```
dreun_na       [dreun]        normal              verb(hebben,sg1,part_intransitive(na))
dreun_op       [dreun]        normal              verb(hebben,sg1,part_sbar(op))
dreun_op       [dreun]        normal              verb(hebben,sg1,part_transitive(op))
2 |: * De beer is los
top_features=grammar
parse: De beer is los
Lexical analysis: 4 words, 243 -> 8 -> 8 -> 8 tags, 42 signs, 170 msec
Parsed (70 msec)
created object: 1
[De beer is los]
Q#undefined|De beer is los|1|1|-0.05113238180000002
created object: 2
[De beer is los]
Q#undefined-2|De beer is los|1|1|-0.02903353880000002
cputime total 260 msec
Found 2 solution(s)
32 history items
             top=top
                |
            -- =smain
        _____________________
       |           |        |
     su=np      hd=verb predc=adj
    ________
   |       |       |        |
det=det hd=noun   ben      los
   |       |
  de     beer
3 |: quit
cramer@rana:~>
```

- Rather than using the [@postag] technique described above, it might be that a certain corpus has multiple occurrences of a word that Alpino does not know about. In that case, you can add the word to the lexicon interactively (it remains in the lexicon for this particular session of Alpino only). Using the command interpreter, the command:

```
add_lex curly mooi
```

tells the Alpino lexicon that the word `curly` has the same lexical categories as the word `mooi`. It also works for multiple word units:

```
add_lex body mass index index
```

POS-tagging

One very important step in syntactic annotation is POS-tagging: determining which word group a word belongs to. Having precise knowledge about these categories can speed up the annotation process considerably, because then you are able to prune large parts of the search space manually.

Almost all words have several POS-tags. A word can be ambiguous because it belongs to different main categories (*werk* can be either a verb or a noun), but also because they differ in details (*geven* can be either intransitive, transitive or ditransitive, some words have other complements as verb phrases, etc.).

In this documents, the main categories and the most frequently used options will be reviewed, as these are most important for the normal user. The very details of some exotic words will thus be left out.

Verbs

The tag of verbs takes at least three arguments:

- The auxiliary verb, when the verb is inflected to a participium (*hebben* or *zijn*);
- The tense and congruent information (*sg1*, *sg3*, *pl*, *past(sg)*, *past(pl)*, *psp*, *inf*);
- The possible complements.

For the latter, there is a large variety of possibilities. These are the basic ones:

Table 1. Verb's fourth argument, single forms

Tag	Example
intransitive	Ik *schrijf*
transitive	Ik *schrijf* een brief
np_np	Ik *schrijf* mijn moeder een brief
so_pp_np	Ik *schrijf* een brief aan mijn moeder
so_np	Ik *schrijf* mijn moeder
refl	Ik *schaam* me
copula	Ik *ben* een echte fietser
pred_np	Ik *vind* dat leuk
pc_pp(over)	Ik *schrijf* over de ramp
ld_pp	Ik *zwaai* naar mijn moeder
er_er	Er *zijn* er die voetballen leuk vinden
part_intransitive(mee)	Ik *schrijf* mee
fixed([svp_pp(op,naam),acc],norm_passive)	Ik *schrijf* die overwinning op mijn naam
vp	Ik *speel* om te winnen
sbar	Ik *schrijf* dat ik huil
cleft	Het *was* op Piet dat hij een uur moest wachten
aan_het	Ik *ben* aan het lopen
aci	Ik *laat* hem fietsen
aux_psp_hebben	Ik *heb* gefietst
aux_psp_zijn	Ik *ben* nog een uurtje gebleven
aux(inf)	Ik *zal* fietsen
aux(te)	Ik *blijk* te blozen
passive	Ik *word/ben* door mijn moeder uitgezwaaid
te_passive	Ik *ben* te vinden in de kelder
dip_sbar_subj	Ik *denk* , zei Piet , dat hij komt

With a bit of creativity one can easily see the meaning of related POS-tags. A few examples are:

Table 2. Verb's fourth argument, combined forms

Tag	Example
ld_adv	Ik *ben* daar
ld_dir	Ik *loop* huiswaarts
	Ik *loop* het bos in
np_ld_pp	Ik *schop* de bal naar het doel
np_ld_dir	Ik *duw* hem huiswaarts
	Ik *stuur* hem het bos in
sbar_subj	Het *lijkt* dat je geschikt bent
sbar_subj_so_np	Het *lijkt* mij dat je geschikt bent
refl_sbar	Ik *schaam* me dat ik dat gedaan heb
copula_sbar	Het *is* leuk dat je fietst
copula_vp	Het *is* leuk om te fietsen
so_copula	Ik *ben* hem trouw
np_aan_het	Ik *heb* de motor aan het lopen
np_sbar	Ik *schrijf* mijn moeder dat ik huil
er_pp_sbar(achter)	Ik *ben* erachter dat je me bedriegt
pred_np_sbar	Ik *vind* het leuk dat je bloost
pred_np_vp	Ik *vind* het leuk om te blozen
pred_pc_pp(van)	Ik *word* verlegen van jouw aandacht
part_transitive(aan)	Ik *schrijf* een fonds aan
part_np_np(voor)	Ik *schrijf* hem een medicijn voor
fixed([[het,apezuur],refl],no_passive)	Ik *schrijf* me het apezuur
fixed([{[acc(gesprek),pc(met)]}],no_passive)	Ik *heb* een gesprek met de rector
aux_modifier(inf)	Het moet raar lopen , *wil* hij komen

Particles

There are no attributes related to particles (separable verb prefixes). They are just as they are. Particles are used as separable verb prefixes as `op` in `Hij belde ons op`, and also as a post-position in a prepositional phrase, such as `af` in `van het begin af had ik twijfels`.

Fixed

The category *fixed* is used for the frozen part of verbal idioms. Examples of these are *ter dood*, *in de koude kleren* and *uit het oog*.

Nouns and related tags

Just as verbs, all noun tags have three compulsory arguments:

- Its determiner (*de*,*het*,*both*);
- Whether it is countable (*count*, *mass*);
- The number of the noun (*sg*, *pl*, *both*, *meas*).

The special value `meas` is used for singular measure nouns that combine nonetheless with plural numbers (`dertien jaar`, `twintig meter`).

The optional fourth attribute of nouns shows which complement(s) the noun takes:

Table 3. Noun's fourth argument

Tag	Example
start_app_measure	onder het *motto* : we zullen wel zien
app_measure	de stof *insuline*
np_app_measure	op *camping* de Mokerhei
measure	een *plakje* words
sbar	het idee dat je kunt vliegen
vp	het idee om te gaan vliegen
subject_sbar	het is een *feit* dat hij komt
subject_vp	het is een *drama* om te moeten wachten
pred_pp(van)	dat is van groot *belang*

Some of these distinctions are somewhat subtle. The label `app_measure` is used if the following noun is understood somewhat like a name. In many cases, you can add the word *genaamd* (entitled): *een stof genaamd insuline*. On the other hand, the `measure` category is used for words such as *plakje, kopje, hoeveelheid* which often indicate a quantity. The `np_app_measure` is used for words that take a full NP modifier (typically including a determiner), often headed by a proper name.

The category `pred_pp(Prep)` is used for nouns which, if combined with the specified preposition Prep, form a predicative PP complement.

The `subject_sbar` and and `subject_vp` tags indicate that the noun occurs as a predicative complement with a sentential subject (sbar or vp).

Verbal nouns

Verbal nouns are verbs that have the syntactical function of a noun (nominalizations). An example is: *Ik heb het fietsen nog niet afgeleerd*. They have the same complements as their corresponding verb entries in the lexicon. An example of this is: *Het in Utrecht wonen bevalt hem* where *in Utrecht* is an LD complement of *wonen*.

Determiner

Determiners are divided in different categories: gen_determiner, comp_determiner, name_determiners, tmp_determiner and regular determiners.

Regular determiners can be divided in a few categories, and they all take a different set of arguments. The type of the determiner is expressed in the first argument. An overview of all types can be found in the table. Besides the first argument, determiners can have up to five additional arguments, depending on the type of the determiner:

- nwh/wh: this indicates whether or not a wh-question can be formed with this word
- nmod/mod: this indicates whether or not the determiner can be modified. An example of such modification is *geen* in *bijna geen*.
- pro: this indicates that the determiner can function as a pronoun.
- yparg/nparg: this binary feature indicates whether the determiner, if used as a pronoun, can occur as a complement of a preposition. This distinction captures *met dit* / *hiermee* pattern.
- wkpro/nwkpro. This indicates whether the determiner, if it is used as a pronoun, is a weak pronoun or not. There are several constraints on the syntactic behaviour of weak pronouns (they cannot take modifiers/complements, they cannot occur in topic, etc.). This feature indicates that the determiner can combine with the post-np adverb *meer* in sentences *geen boeken meer weinig - voorraad meer*.

Table 4. Minor determiner types

Tag	Example
determiner(de)	*De* jongen is flink
determiner(deze,nwh,nmod,pro,yparg)	*Deze* jongen is flink
determiner(alle,nwh,mod,pro,nparg)	*Alle* jongens zijn flink
determiner(geen,nwh,mod,pro,yparg,nwkpro,geen)	*Geeneen* jongen is flink
determiner(wat)	*Tal van* jongens zijn flink
determiner(wat,nwh,mod,pro,yparg)	*Genoeg* jongens zijn flink
determiner(elke,nwh,mod)	*Iedere* jongen is flink
determiner(pron)	*Jouw* jongens zijn flink
determiner(sg_num)	*Menig* jongen is flink
determiner(pl_num,nwh,nmod,pro,yparg)	*Meerdere* jongens zijn flink
determiner(welke)	*Enige* jongens zijn flink
determiner(zulke)	*Zulke* jongens zijn flink

Pronouns

Pronouns have a lot of arguments in Alpino. Because most of the pronouns are not ambiguous, this is only needed for the engine underneath, and is therefore less important for the back-end user of the system. Still, we would like to show a few types of pronouns in their usual contexts.

Table 5. Pronoun types

Tag	Example
pronoun(nwh,thi,pl,de,both,indef)	*Beide* gingen naar school
pronoun(ywh,thi,both,de,both,indef)	*Wie* is dat ?
rel_pronoun(de,no_obl)	De jongen *die* ik bekeek , keek naar me
reflexive(je,both)	Waar houd je *je* mee bezig ?

Adjectives

The first argument shows a list in which forms an adjective can occur:

Table 6. Adjective's first argument

Tag	Example
no_e	de auto is *mooi*
e	de *mooie* auto
both	het *malafide* bedrijf; het bedrijf is *malafide*
pred (only functions as predicative)	de kast staat *ondersteboven*, but not: de *ondersteboven*e kast
er, ere, st, ste (comparative forms)	*mooier*, *mooiere*, *mooist*, *mooiste*
ende/end (gerund with/without e)	een auto *komende* uit de richting Assen
postn	de jongen *afkomstig* uit Palestina

The second argument shows whether the adjective can behave as an adverb as well:

Table 7. Adjective's further attributes

Tag	Example
adv (can be used as vp modifier)	De jongen loopt *mooi*
nonadv (cannot be used as vp mod)	*De jongen loopt *nederlandstalig*
padv (can be used as predm, but not vp mod)	Hij loopt *herkenbaar* over straat
both (can be used both as predm and vp mod)	Gek van vreugde liep hij rustig weg
	Hij heeft deze zaken gek aangepakt
locadv (can be used as locative vp mod)	De jongen loopt *ver*
tmpadv (can be used as temporal vp mod)	Hij heeft *kort* geaarzeld

A further (optional) argument shows which complements the adjective can have. These are very similar to the complements for verbs.

Prepositions

The preposition tag typically has one argument, which is a list (often the empty list). The list containts potential post-positions, with which the current preposition can combine to form a hd/obj1/hdf triples.

In some cases there is a further second argument for prepositions which indicate more specialized uses, as documented in this table:

Table 8. Types of prepositions

Tag	Example
preposition(over,[])	Het leven gaat niet *over* rozen
preposition(achter,[aan,door,langs,om,uit])	*achter* de dief aan
preposition(op,[],pc_adv)	Ik rekende *op* vanmiddag
preposition(ondanks,[],sbar)	Ik kom *ondanks* dat ik me slecht voelde
preposition(ongeacht,[],of_sbar)	Ik kom *ongeacht* of ik me slecht voel
preposition(van,[],pp)	Een boek *van* voor de oorlog
preposition(van,[],loc_adv)	Ik kom *van* boven
preposition(sinds,[],tmp_adv)	Hij woont hier *sinds* gisteren
preposition(tegenin,[],extracted_np)	Hij gaat er niet *tegenin*
preposition(te,[],nodet)	Hij woont *te* Assen
preposition(als,[],pred)	Dat beschouw ik *als* ongepast
preposition(voor,[],voor_pred)	Ik houd het *voor* onmogelijk dat je komt
preposition(met,[],absolute)	Met Piet achter het stuur zijn we verloren

Adverbs

Table 9. Types of adverbs

Tag	Example
adverb(reeds)	Op woensdag wist zij dat *reeds*
modal_adverb(reeds)	*Reeds* op woensdag wist de minister dat
postnp_adverb(reeds)	Deze weeks *reeds* wist de minister dat
postadv_adverb(reeds)	Gisteren *reeds* wist de minister dat
post_wh_adverb([dan,ook])	Zij kunnen waar *dan ook* over de wetenschap praten
wh_adverb(hoezo)	*Hoezo* wist de minister dat al ?
waar_adverb(waarover, over)	*Waarover* praten jullie ? (*Waarover* becomes a pc)

Tag	Example
er_adverb(daarmee, met)	Hou daarmee op ! (*Daarmee* -> pc)
sentence_adverb(bovendien)	*Bovendien* is de rente weer op een acceptabel niveau
loc_adverb(onderaan)	Groningen bungelt *onderaan* (*onderaan* -> ld)
dir_adverb(bergop)	De trend van de laatste jaren is duidelijk *bergop* (*bergop* -> ld)
tmp_adverb(gisteren)	*Gisteren* wist de minister dat al
predm_adverb([en,masse])	Ze gingen *en masse* Berlusconi stemmen

The difference between a sentence_adverb and an ordinary adverb is, that the distribution of sentence_adverbs is more limited.

Complementizers

Table 10. Types of complementizers

Tag	Example
complementizer (will become comp)	*Omdat* de fresco's mooi waren viel het bezoek niet tegen
complementizer(sat) (will become comp)	*Behalve* dat de fresco's mooi waren , was er niet veel te zien
	Behalve de mooie fresco's was er niet veel te zien
complementizer(root) (will become dlink)	*Maar* het bezoek viel tegen
	Maar omdat het bezoek tegenviel …

Conjunctions

Table 11. Types of conjunctions

Tag	Example
conj(en)	Ik zag hem *en* groette hem
conj(alias)	Ik ontmoette Johan V. *alias* de Hakkelaar
left_conj(maar_ook), right_conj(maar_ook)	*Niet alleen* de motoren *maar ook* de auto's gingen op de bon

UNKNOWN words

UNKNOWN words are wordt that cannot be placed in any context, and will be entirely disregarded in the parsing process. This typically occurs only for parts of multi-word-units.

Proper names

Alpino contains long lists of names, as well as a number of heuristics to recognize names. The tag `proper_name` is used for these cases. The sub-features include agreemnt (often left unspecified), and a further named entity class such as PER (person names), LOC (geographical names) and ORG (organization names). For annotation purposes you can safely ignore this distinction.