Alpino User Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Alpino is a collection of tools and programs for parsing Dutch sentences into dependency structures.

Chapter 1

Downloading and Installing

The binary distribution is available only for some platforms (specific requirements are listed below). Installing involves the following steps:

1. Download the relevant file for your architecture at Alpino download site.
2. Extract the contents of that file in a place of your liking, e.g., in your home-directory, and do the following:

   ```
   tar xzf Alpino-bla-bla-19327.tar.gz
   ```

The binary file depends on the Tcl/Tk libraries as well as a number of system libraries. These libraries are often available for a standard Linux distribution. A few likely candidates for trouble are included in the distribution (under create_bin/extralibs)

In addition, some of the tools and programs that are shipped with Alpino require additional packages. Here is the beginning of a list. Typically you need a recent version ;-)

- go
- tcl/tk
- lxml
- python2
- python2 library lxml
- dictzip (sometimes packaged with dictd)
- libpopt
- libz

Chapter 2

Running Alpino

In other that you can use Alpino and the related tools, you need to define your ALPINO_HOME environment variable. This variable should point to the directory that containts the Alpino stuff. A typical way tot do this, is to add the following lines to the file .bashrc in your home directory, assuming you use bash, and assuming you extracted the Alpino stuff in a directory called Alpino in your home directory:

```
export ALPINO_HOME=$HOME/Alpino
export PATH=$PATH:$ALPINO_HOME/bin
```

2.1 Basic Usage

The various usages of Alpino can be summarized as follows:

```
Alpino [Options]
Alpino [Options] -parse
Alpino [Options] -parse W1 .. Wn
```

In the first form, the system will work in interactive mode. By default, it will start the graphical user interface. Use the options -tk or -notk to enable/disable the gui explicitly.

In interactive mode, you are in effect using the Hdrug command interpreter and (optionally) the Hdrug graphical user interface. Refer to the Hdrug documentation for more information.

In the second form, the system will read lines from standard input. Each line is then parsed, and various output is produced (the actual output depends on the various options that are set). For those with less experience with Unix, this form allows you to parse larger collections of sentences using input redirection:

```
cat MySentences.txt | Alpino -parse
```

Finally, in the third form, the system will parse the single sentence given on the command line, provide output (again, depending on the options), and quit. In this form, the input sentence is given as a sequence of tokens W1 .. Wn on the command line.

2.1.1 Options

There are a whole bunch of options which affect the way in which the system runs. For most options, there is documentation available from the graphical user interface. In order to have access to this, run:

```
Alpino
```

Some frequently used options are given here.

Flag=Value
Assigns Value to global variable Flag. See below for a list of flags with suggested values.

The remaining options all start with a dash:

-flag Flag Value
Similar to the above. The difference is that in the Flag=Value syntax, the Value is parsed as a Prolog term. In the -flag Flag Value syntax, on the other hand, the Value is parsed as a Prolog atom. In many cases this does not make a difference. Remember: for path names, you need the second format.

-tk
Use the graphical user interface. Only makes sense for interactive operation.

-notk
Do not use the graphical user interface.

-cmd Goal
Evaluates Prolog Goal; Goal is parsed as Prolog term.

-l File
Loads the Prolog file, using the Prolog goal use_module(File).

-slow
The alternative to the *veryfast*-option. The system will find all possible parses for the input. No part-of-speech pre-processor is applied. No beams are used in search. The parser does not enforce the "guides" technique.

-veryfast
This option uses a set of options to improve the speed of the parser. In this mode, the parser only delivers the first (best) analysis. The part-of-speech pre-processor is applied. Guided parsing is applied. Several heuristics apply to limit the search space.

2.2 Flags (global variables)

There are (way too) many global variables (called flags) which will alter the behavior of Alpino. These flags typically already have a default value. The value can be changed using command line options. The values can also be changed in interactive mode using either the command interpreter or the graphical user interface. In the graphical user interface, documentation for each option is provided. Boolean flags have one of the values "on" or "off". The following options appear most relevant:

debug
Integer (typically 0, 1 or 2) which determines the number and detail of debug and continuation messages.

demo
Boolean flag which determines if a visual impression of the parse is produced (either to standard output, or the graphical interface if it is running).

disambiguation_beam
Integer which determines during unpacking the how many best analyses are kept for each ‘maximal projection’. A larger value will imply slower and more accurate processing. The value 0 is special: in that case the system performs a full search (hence maximal accuracy and minimal speed). The value of this flag is ignored in case unpack_bestfirst=off.

disambiguation_candidates_beam
Integer which determines during parsing how many parses are produced for any given maximal projection. From these, only the best are kept for further processing later (using the disambiguation_beam flag). This flag can be used to limit the number of parses that are computed in the first place. A value of 0 means that all parses are produced. If the value is N>0, then only the first N parses are computed.

end_hook
Perhaps the most important flag is the end_hook flag which determines what the system should do with a parse that it has found. Typically this involves printing certain information concerning the parse to standard error, standard output or a file. Various examples are provided below.

first_line_no
Integer which you can use to enforce that Alpino first ignores a number of lines, and then starts parsing after that many lines. Normally, the value of this flag will be 0 (parse all lines), but in some cases it is useful to ignore the first part of an input file (because perhaps those lines already were parsed in an earlier session).

min_sentence_length
Integer which determines minimum sentence length. Sentences with less words are ignored. This flag is undefined by default.

max_sentence_length
Integer which determines maximum sentence length. Sentences with more words are ignored. This flag is undefined by default.

number_analyses
This flag determines the number of analyses that is passed on by the robustness / disambiguation component. If the value is 0, then the system simply finds all solutions.

parse_candidates_beam
Integer which determines during parsing (first phase) how many sub-parses are produced for any given sub-goal. A value of 0 means that no limit on this number is enforced.

pos_tagger
Boolean flag to determine whether to use a POS-tagger to filter the result of lexical lookup. The POS-tagger is based on unigram, bigram and trigram frequencies. This filter can be made more or less strict using the pos_tagger_n flag.

pos_tagger_n, pos_tagger_p, pos_tagger_m
These flags take numerical values which determines how and how much filtering should be done by the POS-tagger which filters the result of lexical lookup (if pos_tagger=on). Refer to the source code for details. Default values should give good performance.

user_max
This flag takes a numberical value which is then used as the maximum number of milliseconds that the system is given for each sentence. So, if the value is set to 60000, then if a parse for a given sentence requires more than a minute of CPU-time, that parse is aborted. Because the system can sometimes spend a very long time on a single (long, very ambiguous) sentence, it is often a good idea to use this time-out.

xml_format_frame
Boolean flag that indicates whether the system should produce verbose xml output (containing various detailed lexical features).

2.3 Examples with end_hook

2.3.1 xml

Suppose you want to save the dependency structure of the best parse of each sentence as xml. In that case, the following command might be what you want:

```
Alpino -flag treebank $HOME/tmp end_hook=xml -parse
```

For each sentence read from standard input, an xml file will be created in the directory $HOME/tmp containing the dependency structure. The files are named 1.xml, 2.xml, ... etc (if you want to start counting from N+1, then you can use the current_ref=N flag to initialize the counter at N).

For browsing and querying such xml files, check out the scripts that come with the Alpino Treebank.

2.3.2 dependency triples

Rather than a full dependency structure for each sentence, it might be that you only care for the dependency triples (two words, and the name of the dependency relation). For this, you might try:

```
Alpino end_hook=triples -parse
```

For an input consisting of the three lines:

```
ik houd van spruitjes
ik ben gek op spruitjes
Mijn minimal brain dysfunction speelt weer op
```

this writes out:

```
houd/[1,2]|su|ik/[0,1]|1
houd/[1,2]|pc|van/[2,3]|1
van/[2,3]|obj1|spruitje/[3,4]|1
ben/[1,2]|su|ik/[0,1]|2
ben/[1,2]|predc|gek/[2,3]|2
gek/[2,3]|pc|op/[3,4]|2
op/[3,4]|obj1|spruitje/[4,5]|2
speel_op/[4,5]|su|minimal brain dysfunction/[1,4]|3
minimal brain dysfunction/[1,4]|det|mijn/[0,1]|3
speel_op/[4,5]|mod|weer/[5,6]|3
speel_op/[4,5]|svp|op/[6,7]|3
```

Each line is a single dependency triple. The line contains four fields separated by the | character. The first field is the head word, the second field is the dependency name, the third field is the dependent word, and the fourth field is the sentence number. The words are representend as Stem/[Start,End] where Stem is the stem of the word, and Start and End are string positions. If you also want two additional fields with the POS-tags of each word, then use the end_hook=triples_with_frames option. Typical output then looks like:

```
houd/[1,2]|verb|mod|niet/[2,3]|adv|1
houd/[1,2]|verb|pc|van/[3,4]|prep|1
houd/[1,2]|verb|su|ik/[0,1]|pron|1
van/[3,4]|prep|obj1|smurfen/[4,5]|noun|1
```

2.3.3 syntactic structure

Perhaps your interest lies in the syntactic structure assigned by Alpino. The following writes out the syntactic structure for each parse that Alpino finds:

Alpino number_analyses=1000 end_hook=syntax -parse

Each parse is written as a bracketed string, where each opening bracket is followed by the category (preceded by the ampersand). For the phrase

```
de leuke en vervelende kinderen
```

the (rather verbose) output is, where each line starts with the sentence number, the separation character | and the annotated sentence:

```
1| [ @top_cat [ @start [ @max [ @np [ @det de ] [ @n [ @a [ @a leuke ] [ @clist
  [ @optpunct ] [ @conj en ] [ @a vervelende ] ] ] [ @n kinderen ] ] ]
  ] ] [ @optpunct ] ]
1| [ @top_cat [ @start [ @max [ @np [ @det de ] [ @n [ @n [ @a leuke ] ] [ @clist
  [ @optpunct ] [ @conj en ] [ @n [ @a vervelende ] [ @n kinderen ] ] ] ] ]
  ] ] [ @optpunct ] ]
1| [ @top_cat [ @start [ @max [ @np [ @np [ @det de ] [ @n [ @a leuke ] ] ]
  [ @clist [ @optpunct ] [ @conj en ] [ @np [ @n [ @a vervelende ] [ @n
  kinderen ] ] ] ] ] ] ] [ @optpunct ] ]
```

2.3.4 POS-tags

If you only care for the part-of-speech tags that Alpino used to derive the best parse, then you can use the following command:

```
Alpino end_hook=postags -parse
```

In that case, the system prints lines like the following:

```
101|met|0|1|VZ(init)|met
101|de|1|2|LID(bep,stan,rest)|de
101|trein|2|3|N(soort,ev,basis,zijd,stan)|trein
101|mee|3|4|VZ(fin)|mee
101|wil|4|5|WW(pv,tgw,ev)|willen
101|ik|5|6|VNW(pers,pron,nomin,vol,1,ev)|ik
101|vandaag|6|7|BW()|vandaag
```

The fields are seperated by vertical bar and represent the key, the word, begin position, end position, part-of-speech label and lemma.

2.3.5 Internal POS-tags

If you only care for the part-of-speech tags that Alpino used to derive the best parse, then you can use the following command:

```
Alpino end_hook=frames -parse
```

In that case, the system prints lines like the following to standard error (!). For historical reasons, the output goes to standard error, not to standard output.

```
ik|pronoun(nwh,fir,sg,de,nom,def)|1|0|1|normal|pre
vertrek|verb(zijn,sg1,intransitive)|1|1|2|normal|post
houdt|verb(hebben,sg3,pc_pp(van))|2|1|2|normal|post
van|preposition(van,[af,uit,vandaan,[af,aan]])|2|2|3|normal|post
Marietje|proper_name(both)|2|3|4|name(not_begin)|post
```

Each line represents the information of a word. The fields represent: the word, the internal part-of-speech tag, the sentence key, the begin position of the word, the end position of the word, and two further fields that you probably want to ignore.

Note that this type of output provides the internal word categories. If you need the "official" CGN/Lassy-style part-of-speech labels, you need the end_hook "postags" described above.

It is fairly easy to define further output formats. Please contact the author if you have specific requests.

2.4 If something goes wrong

If for some reason Alpino does not work, the first thing you should try is to alter in the Alpino initialization script the line

```
debug=0
```

into

```
debug=1 (or even a higher number)
```

If you are lucky this implies that you get told what is going wrong.

If this doesn't help, send a bug-report to the author. Please.

2.5 Tokenization

The sentences that you give to Alpino are assumed to be already tokenized: a single sentence on a line, and each token (word) separated by a single space. Refer to the section Alpino Tokenizer below for a sentence splitting and tokenization tool. Alpino supports the UTF-8 encoding. Here are some properly tokenized sentences:

```
Dat is nog niet duidelijk . '
We zien hier niet het breken van tandenstokers .
Dodelijke wapens worden gesloopt . '
Slechts enkele nieuwe documenten zijn aan het licht gekomen . '
Er is nog geen bewijs van verboden activiteiten gevonden .
```

If you want to assign an identifier for each sentence, then you write the identifier in front of the sentence, with a | separation character. For instance:

```
volkskrant20030308_12|Dat is nog niet duidelijk . '
volkskrant20030308_14|We zien hier niet het breken van tandenstokers .
volkskrant20030308_15|Dodelijke wapens worden gesloopt . '
volkskrant20030308_17|Slechts enkele nieuwe documenten zijn aan het licht gekomen . '
volkskrant20030308_20|Er is nog geen bewijs van verboden activiteiten gevonden . '
```

It follows that the vertical bar is a special character that is supposed to be not part of the sentence in case there is not an identifier.

Lines which start with a percentage sign are ignored: the percentage sign is used to introduce comments:

```
volkskrant20030308_12|Dat is nog niet duidelijk . '
volkskrant20030308_14|We zien hier niet het breken van tandenstokers .
%% the next sentence is easy:
volkskrant20030308_15|Dodelijke wapens worden gesloopt . '
volkskrant20030308_17|Slechts enkele nieuwe documenten zijn aan het licht gekomen . '
volkskrant20030308_20|Er is nog geen bewijs van verboden activiteiten gevonden . '
```

If you have corpus material that has not been tokenized yet, then you can also use Alpino to tokenize your input. Use the flag *assume_input_is_tokenized* for this purpose. If the value is *on*, then Alpino won't try to tokenize the input. If the value is *off*, it will tokenize the input. It will not do any sentence splitting.

However, note that if you use the built-in Alpino tokenizer, then you cannot use the meta-notation introduced below in section *Bracketed Input

2.5.1 Sentence identifiers

The output of Alpino uses identifiers. For instance, if the end_hook=xml option has been set, the names of the XML files are constructed out of these identifiers. The sentence identifier is either defined explicitly as part of the input sentence, or it is set implicitly by Alpino (simply counting the sentences of a specific session starting from 1). If there are multiple outputs for a given sentence, for instance if you have defined the option number_analyses=10, then each of the results is indexed with the rank of the analysis. The resulting identifier then consists of the sentence identifier and the rank of the analysis, separated by a dash. Example:

```
echo "example1|de mannen kussen de vrouwen" |\
Alpino -notk number_analyses=0 end_hook=frames -parse
```

This yields:

```
de|determiner(de)|example1|0|1|normal(normal)|pre|de|0|1
mannen|meas_mod_noun(de,count,pl)|example1|1|2|normal(normal)|pre|man|0|1
kussen|verb(hebben,pl,transitive)|example1|2|3|normal(normal)|post|kus|0|1
de|determiner(de)|example1|3|4|normal(normal)|post|de|0|1
vrouwen|noun(de,count,pl)|example1|4|5|normal(normal)|post|vrouw|0|1
de|determiner(de)|example1-2|0|1|normal(normal)|pre|de|0|1
mannen|meas_mod_noun(de,count,pl)|example1-2|1|2|normal(normal)|pre|man|0|1
```

```
kussen|verb(hebben,pl,transitive)|example1-2|2|3|normal(normal)|post|kus|0|1
de|determiner(de)|example1-2|3|4|normal(normal)|post|de|0|1
vrouwen|noun(de,count,pl)|example1-2|4|5|normal(normal)|post|vrouw|0|1
de|determiner(de)|example1-3|0|1|normal(normal)|pre|de|0|1
mannen|meas_mod_noun(de,count,pl)|example1-3|1|2|normal(normal)|pre|man|0|1
kussen|verb(hebben,pl,intransitive)|example1-3|2|3|normal(normal)|post|kus|0|1
de|determiner(de)|example1-3|3|4|normal(normal)|post|de|0|1
vrouwen|noun(de,count,pl)|example1-3|4|5|normal(normal)|post|vrouw|0|1
```

NB: there may be a problem with sentence identifiers that contain non-ascii characters. See the section "Known Issues" below.

2.6 Special symbols in the input

The following symbols which might occur in input lines are special for Alpino:

```
| [ ] % ^P
```

The vertical bar | is used to separate the sentence key from the sentence. Only the first occurrence of a vertical bar on a given line is special.

The square brackets are used for annotating sentences with syntactic information. This is described below.

The percentage sign is only special if it is the first character of a line. In that case the line is treated as a comment, and is ignored.

A line only consisting of ^P (control-P) is treated as an instruction to escape to the Prolog command-line interface. This is probably only useful for interactive usage. Typing "halt." to the Prolog command-line reader will return to the state before control-P was typed.

If you need to parse a line starting with a %-sign, then the easiest solution is to use a key (for interactive usage, perhaps use the empty key):

```
3|% is een procentteken
|% is een procentteken
```

The same trick can be used to interpret the vertical bar literally:

```
4|ik bereken P(x|y)
|ik bereken P(x|y)
```

Note that it is currently not possible to use keys that start with the symbol % or |.

If you have square brackets in your input, you need to escape these using the backslash.

2.7 Bracketed input

Alpino supports brackets in the input to indicate:

- syntactic structure
  - @cat
- lexical assignment
  - @postag
  - @posflt
  - @alt (@add_lex in older versions)
  - @mwu

  - @folia
- requirement to skip words
  - @skip
- include phantom words
  - @phantom
- pass on identifier of words

Warning: bracketed input is not supported in case Alpino tokenizes the input sentences on the fly.

A large part of the computational time is spent on finding the correct constituents. For interactive annotation, it is possible to give hints to Alpino about the correct constituent structure 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). The ability to indicate syntactic structure is a nice feature especially for attaching modifiers at the correct location, enumerations and complex nestings. 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 ] , bladmuziek , folders , brochures ] om in
  gesproken vorm .
[ @np De conferentie die betrekking heeft op ondersteunende
  technologie voor gehandicapten in het algemeen ] , bood [ @np een goed
  platform [ om duidelijk te maken hoe uitgevers zelf toegankelijke
  structuren in hun informatie kunnen aanbrengen ] ] .
```

This mechanism can also be used (somewhat unexpectedly perhaps) to indicate what the fragments of the input are, if an input contains several fragments but is not a syntactic unit (this occurs frequently for spoken input, for instance). Suppose your input sentence is the sentence "de boer als hij gepresseerd is hè hij gaat voort in het veld". You can indicate the fragments using a bracket with a category that does not occur in the grammar, e.g., x:

```
[ @x de boer ] [ @x als hij gepresseerd is ] [ @x hè ] [ @x hij gaat voort in het veld ]
```

or even simply:

```
[ @ de boer ] [ @ als hij gepresseerd is ] [ @ hè ] [ @ hij gaat voort in het veld ]
```

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, the resulting dependency structure may need adjusting with the editor afterwards.

```
Hij heeft een beetje [ @postag adjective(both(nonadv)) curly ] haar .
Op een [ @postag adjective(e) mgooie ] dag gingen ze fietsen .
Mijn [ @postag proper_name(sg,'MISC') body mass index ] laat te wensen over .
```

As a variant of the @postag annotation, you can also use the @posflt annotation. In this case, only the functor of the POS-tag is given, and Alpino will allow any lexical assignments which have the same functor. Example:

```
ik wil [ @posflt preposition naar ] huis
ik voel me [ @posflt adjective naar ]
```

Note that the @postag annotation differs from the @posflt annotation, in that in the first case, a lexical category is assigned without consultation of the dictionary. In the latter case, a subset of the lexical categories assigned by Alpino is allowed (this subset can be empty).

The @alt meta annotation can be used to tell the parser to treat a given word in the input with the lexical categories associated with another word. Example:

```
ik wil [ @alt naar naa ] huis
ik voel me [ @alt goed gooooooeeeeed ]
```

In this case, the word *naa* in the input will be treated in lexical lookup as if the word *naar* was given. Similarly, the lexical categories associated with *gooooooeeeeed* are those of the word *goed*.

In older versions, we used (for historical reasons) the label "add_lex" instead of alt. Furthermore, add_lex would also support treating more than a single word as a unit. This can be accomplished using "alt" in combination with skips. For example:

old:

```
ik wil [ @add_lex daarom daar om ] naar huis
```

new:

```
ik wil [ @alt daarom daar ] [ @skip om ] naar huis
```

A further meta-label for lexical assignment is the @mwu label. This indicates that the bracketed sequence of words should be regarded as a single lexical category. The effect is, that lexical assignments that do not respect this bracketing are removed. In case there is no lexical assignment for the words marked @mwu, then it is assumed that this sequence should be analysed as a name.

```
Dat was de periode van het [ @mwu Derde Rijk ]
Op [ @mwu 1 januari 2001 ]
```

Skips. 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.

The @folia meta-notation can be used to inform Alpino about the lemma and postag of the given word. Note that in this case, Alpino will **not** attempt to create an analysis that is consistent with the given lemma and postag, but it will simply pass on the given postag and lemma values to the @postag and @lemma XML attributes of the corresponding node in the XML output. Examples:

```
[ @folia subtiel ADJ(vrij,basis,zonder) Subtiel ]
[ @folia zijn WW(pv,tgw,mv) Zijn ] [ @folia cafévriend N(soort,mv,basis) cafévrienden ] [  ↩
    @folia bekijken WW(pv,tgw,mv) bekijken ] [ @folia hem VNW(pers,pron,obl,vol,3,ev,masc)  ↩
    hem ] [ @folia argwanend ADJ(vrij,basis,zonder) argwanend ] [ @folia . LET() . ]
```

There also is the @id meta-notation. This notation can be used to add a wordid attribute in the XML output of a particular word. In order that this meta-notation can be used in combination with other, the notation **precedes** the word for which it is meant.

```
[ @id w1 ] ik [ @id w2 ] besta
[ @id w1 ] [ @folia subtiel ADJ(vrij,basis,zonder) Subtiel ] was het niet
```

Chapter 3

Alpino Hooks

Note that Alpino currently is not really implemented as a Prolog library. Yet, you might find it useful to know that you can call the Alpino parser from Prolog using the following two predicates:

```
alpino_parse_line(Ref,String)
alpino_parse_tokens(Ref,Tokens)
```

Here, Ref is an atomic identifier which is used for administrative purposes, so that results for different sentences can be easily distinguished. In the first case, the input is a string which may need to be tokenized (see discussion above). In the second case, Tokens is a list of atoms where each atom is a token (word, punctuation mark) of the sentence. Here are some examples of the use of these predicates:

```
?- alpino_parse_line(s2323,"ik besta").

?- alpino_parse_tokens(s2324,[ik,besta]).
```

If you want to extend the functionality of Alpino, you may be interested to learn about the following (Prolog-) hooks.

The hook predicate

```
alpino_start_hook(Key,Sentence)
```

is called before the parse of each sentence. Key is instantiated as the atomic identifier of the sentence. Such identifiers are set using the current_ref Hdrug flag.

The hook predicate

```
alpino_result_hook(Key,Sentence,No,Result)
```

is called for each parse. Key and Sentence are as before, No is an integer indicating the number of the parse (parses for a given sentence are numbered from 1 to n), and Result is an internal representation of the full parse itself.

The hook predicate

```
alpino_end_hook(Key,Sentence,Status,NumberOfSolutions)
```

is called after the parser has found all solutions. NumberOfSolutions is an integer indicating the number of solutions. Status is an atom indicating whether parsing was successful, or not. Possible values are the following - their meaning is supposed to be evident:

```
success
failure
time_out
out_of_memory
```

Other values describe more exotic possibilities, please consult the hdrug_status flag in Hdrug for further details.

Chapter 4

Alpino Tokenizer

The Alpino distribution contains a directory Tokenization with various tools that you can use to tokenize your input into the required format for Alpino.

The script "tokenize.sh" does the tokenization. It uses "tok", which is a C program created by the FSA Utilities that does most of the tokenization. This is much more efficient than an equivalent Perl script using regular expressions. In addition there are a number of scripts that do some more difficult stuff that requires more global information (for which no sequential transducer can be defined).

In the input, every paragraph of text is supposed to be given on a single line. If your input is not like that, cf the script paragraph_per_line.

4.1 Example 1

For an example text, cf the text file t1.txt which contains the following:

```
Het PIONIER project Algorithms for Linguistic Processing van Gertjan
van Noord is een onderzoeksproject op het gebied van de computationele
taalkunde.

Een van de taken van de taalkunde is het opstellen van modellen voor
natuurlijke talen. Het model (grammatica) beschrijft bijvoorbeeld de
mogelijke uitingen van het Nederlands en hun bijbehorende
betekenis. Zo'n model beschrijft dus (onder andere) de relatie tussen
vorm en betekenis. Een belangrijke vraag in de computationele
taalkunde is vervolgens hoe je deze relatie algoritmisch kunt
bepalen. Hoe kun je voor een gegeven uiting van het Nederlands de
bijbehorende betekenis berekenen?

Het PIONIER onderzoek richt zich hierbij op twee fundamentele
problemen: ambiguiteit en efficientie. Het probleem van ambiguiteit
doet zich voor in zelfs de meest onschuldige zinnetjes:

    Mijn vader zagen we niet meer

Het is bijzonder lastig (voor een computer) om te voorkomen dat in dit
zinnetje `zagen' wordt begrepen als `zagen met een zaag', terwijl door
mensen deze zin normaal gesproken niet als meerduidig wordt ervaren.

Het tweede probleem behelst de efficientie van de verwerking. Voor de
modellen die tot nu toe worden opgesteld kan worden aangetoond dat de
verwerkingstijd polynomiaal (soms zelfs exponentieel) zou moeten
toenemen bij het toenemen van de lengte van de uiting. Dit is echter
```

```
in tegenspraak met de geobserveerde efficientie van het menselijk
taalvermogen.

Het PIONIER project zal de hypothese onderzoeken (die teruggaat op het
werk van Chomsky in het begin van de zestiger jaren) dat
taalverwerking kan worden gemodelleerd door automaten van een zeer
eenvoudig type (``finite-state automaten'') die een gegeven model van
de taal slechts benaderen. Zo'n aanpak zou onder andere verklaren
waarom uitingen met `center-embedding' en `cross-serial dependencies'
snel onbegrijpelijk worden, maar wel als grammaticaal moeten worden
beschouwd.

Een volledige beschrijving van het voorgestelde onderzoek is
beschikbaar op het Internet www.let.rug.nl/~vannoord/alp/index.html.
```

In order to tokenize your text, you would first ensure that each line contains a paragraph of text with the paragraph_per_line script (this essentially removes single newlines from the input), and then pipe the result to the tokenizer:

```
./paragraph_per_line t1.txt | ./tokenize.sh
```

This results in a file where each line contains a single tokenized sentence.

4.2 Example 2: identifiers

Often it is desirable to have identifiers which indicate the paragraph number and the sentence number in that paragraph. For this purpose, we need the two scripts ./add_key and ./number_sents.

```
./paragraph_per_line t1.txt | ./add_key | ./tokenize.sh | ./number_sents
```

Each line is now prefixed with an identifier X-Y where X is the paragraph number and Y is the sentence number. The first few lines of the result are listed here:

```
1-1|Het PIONIER project Algorithms for Linguistic Processing van Gertjan van Noord is een  ↩
    onderzoeksproject op het gebied van de computationele taalkunde .
2-1|Een van de taken van de taalkunde is het opstellen van modellen voor natuurlijke talen  ↩
    .
2-2|Het model ( grammatica ) beschrijft bijvoorbeeld de mogelijke uitingen van het  ↩
    Nederlands en hun bijbehorende betekenis .
2-3|Zo'n model beschrijft dus ( onder andere ) de relatie tussen vorm en betekenis .
2-4|Een belangrijke vraag in de computationele taalkunde is vervolgens hoe je deze relatie  ↩
    algoritmisch kunt bepalen .
2-5|Hoe kun je voor een gegeven uiting van het Nederlands de bijbehorende betekenis  ↩
    berekenen ?
3-1|Het PIONIER onderzoek richt zich hierbij op twee fundamentele problemen : ambiguiteit  ↩
    en efficientie .
3-2|Het probleem van ambiguiteit doet zich voor in zelfs de meest onschuldige zinnetjes :
```

Chapter 5

Tools for Error Mining

TODO

Chapter 6

Running Alpino as a Server

As of march 2011, there is experimental support for running Alpino as a server program. The server is started using the Prolog command

```
alpino_server
```

Communication with a client proceeds through the use of internet sockets. There are options to set the domain-name and the port number. There is a makefile Makefile.start_server which can be used to simplify starting up an Alpino server. For every request that the server receives, a new client process is started which takes care of that request. An Alpino server only runs under Linux.

The advantage of using a server is, that the Alpino functionality can be made available to machines on which Alpino does not run.

Using the Alpino server to parse a sentence is much faster than initializing a separate Alpino process for that purpose.

Implementing an Alpino client is straightforward in most programming languages. For instance, the following defines a simple Alpino client in Python:

```
import socket

def alpino_parse(sent, host='vingolf.let.rug.nl', port=42424):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect((host,port))
    sent = sent + "\n\n"
    s.sendall(sent.encode())
    total_xml=[]
    while True:
        xml = s.recv(8192)
        if not xml:
            break
        total_xml.append(str(xml,encoding='utf8'))

    return "".join(total_xml)



def main():
    sentence=input("Geef de zin die Alpino moet analyseren: ")
    xml=alpino_parse(sentence)
    print(xml)


if __name__ == '__main__':
    main()
```

The Alpino Server is experimental, and the interface may change in future versions.

Chapter 7

Known Issues

7.1 Parsing strings with odd characters if the GUI is running and you use SICStus

Alpino assumes that your input is in UTF8. There is a problem if you run Alpino with the graphical user interface, and you attempt to parse sentences through the command interpreter (either the Hdrug/Alpino command interpreter, or the standard Prolog command interpreter). In this situation, certain non-ASCII characters are not treated appropriately, due to a bug in the combination of tcl/tk and Sicstus3. Typing in sentences through the parse widget works without any problems though. Typing in sentences through the command interpreter also works without any problems if you did not start the graphical user interface.

This is the feedback from SICS:

This is what happens

1. You have an UTF-8 locale (invoke the locale command to verify this).
2. Inititializing Tcl/Tk will change the process locale from the default "C" locale to whatever locale is inherited from the environment. I.e. when Tcl/Tk is initialized the process locale becomes the UTF-8 locale in (1). Loading library(tcltk) will initialize Tcl/Tk.
3. The SICStus tokenizer classifies non-Latin1 characters using locale sensitive functions (iswpunct(), ...). So when the process locale is UTF-8 the U+2018 and U+2019 in l.pl are classified as symbol characters which, rightfully, gives a syntax error.
4. When library(tcltk) is not loaded then the process will ignore the locale from the environment and use the "C" locale where all code points above 255 are treated as lower case. In this case the U+2018 and U+2019 characters are just treated as any other lower case characters and they are parsed as part of the longer atoms.

So, the SICStus bug is that character classification is affected by the process locale. The Tcl/Tk bug/misfeature is that initializing Tcl/Tk affects the global process locale.

7.2 File-names with non-ascii characters

If you parse sentences, and save the result using the key of that sentence, then if your key includes non-ascii characters the resulting file-name may be wrong. This happens only if you use the SICStus version, the SWI-version treats such keys correctly.

Example:

```
% echo "2-1-π1|Plotseling vlogen ze overeind" | Alpino end_hook=xml -flag treebank . -parse
% ls *.xml
2-1-?1.xml
```