DIPLOMARBEIT

Transformation of natural language into formal logic - possibilities and limits

ausgeführt am Institut für Computersprachen,
Abteilung für Anwendungen der Formalen Logik
der Technischen Universität Wien
in Zusammenarbeit mit der SIEMENS AG Österreich

unter der Anleitung von
AO. Univ. Prof. Dr. phil Alexander Leitsch
und Dr. Peter Hrandek

durch
Andreas Kaltenbrunner
Matr. Nr.: 9526095

1100 Wien, Antonsplatz 13/19

Wien, 28th September 2000

Abstract

The aim of this work is to develop and implement algorithms which transform single English sentences without concern to possible relations to other sentences into equivalent statements in propositional and First-order logic. Contrary to prior works on this subject it should not be limited by a small grammar or vocabulary. The limits of this algorithm will be discussed as well as the general problems of translating language to logic.

Acknowledgements

I wish to thank Dr. Peter Hrandek who gave me the possibility to work on this project in co-operation to Siemens Austria and introduced me into the vast area of language processing and the logic of natural languages, Robert Arinyo of the UPC Barcelona who taught me the logical foundations and created the wish to work on this project and of course Univ. Prof. Dr. Alexander Leitsch for the excellent support. I also have to thank Univ.-Ass. Dr. Gernot Salzer for some useful hints en technical problems with Perl and Prolog.

Contents

Chapter 1

Introduction

In this work we will investigate the possibility and frontiers of automatic transformation of English sentences into statements in propositional logic and first order logic e.g. the Sentence:

> If every man loves a woman and every woman loves a man then everyone loves someone

Should be transformed into:

(every man loves a woman)$\wedge$(every woman loves a man) $\rightarrow$ (everyone loves someone)

and

$$\begin{array}{llllllll} (\forall x_1 & man(x_1) & \rightarrow & \exists x_2 & woman(x_2) & \wedge & loves(x_1, x_2)) & \wedge \\ (\forall x_3 & woman(x_3) & \rightarrow & \exists x_4 & man(x_4) & \wedge & loves(x_3, x_4)) & \rightarrow \\ (\forall x_5 & person(x_1) & \rightarrow & \exists x_6 & person(x_6) & \wedge & loves(x_5, x_6)) & \end{array}$$

We will develop two algorithms and implemente them using PROLOG and PERL. The problem for First-order logic is presented in [8]. In this work the algorithm is also implemented in Prolog and uses the lambda calculus to develop it. But unlike our aim the language it works on is a very small subset of English, and since we Want to me much mire general we will go our own ways in developing the algorithm.
Of course we will need a dictionary to determine the part of speech of every occurring word. By knowing this we can parse the phrase by a phrase structure grammar as shown in [2], [3] and [4].
Given the syntactical structure we transform into Propositional and First-order Logic according to [7] and [1].
We will talk about the frontiers referring to [11] and [10] and finally we will implement the algorithm and see a few examples of its performance.

1.1 Structure of the Thesis

This work is split into 4 chapters:

Chapter 1 gives a short introduction and some general thoughts.

Chapter 2 contains the algorithms, examples showing how they work and some thoughts about how to transform connectors and quantifiers into logic.

Chapter 3 confronts us with the area of problems in simplifying language to logical statements. Especially if this statements are restricted to First-order Logic. It also presents some ideas how to get a bit better algorithms.

Chapter 4 describes the implementation of the algorithms in Prolog and Perl called la2lo, how to install it and shows a few examples, demonstrating its performance.

Appendix contains listings of the modules of la2lo and the grammar it used.

1.2 Before we start

Before we start to develop our algorithms we have to think about some principal things. What we want is to transform any given sentence in English into a logical form that is equivalent, which means its semantic interpretation is equivalent. We will see that there are some principal problems in doing this. We assume the reader to be familiar with grammars, First-order and Propositional Logic and formal languages.
Furthermore we assume to have an idea what to expect if we translate an English statement into logic, since logic is an attempt to determine the truth value of a statement.
In the following we will give a few additional definitions and theorems to the common logical ones.

1.2.1 Horn's Theorem

Contrary to the well known laws of de Morgan for pure logic, there is another law that gives contrary imagination of how to deal with negations in some special linguistic cases.

Horn's Theorem 1.1 $\neg p \wedge \neg q \leftrightarrow \neg(p \wedge q)$

p and q are not totally independent statements like in the case of the laws of de Morgan. We have to imagine them as something like:

Mary and Susan do not speak of love.

Then p would be equivalent to "Mary speaks of love" and q to "Susan speaks of love". The above sentence can be equivalent to

1. Mary speaks of love but Susan does not.
2. Mary does not speak of love and Susan does not speak of love.

Case 1 is the classical logic interpretation which fulfils the laws of de Morgan, whereas the more common interpretation of case 2 leads us to Horn's Theorem. In the next chapter we will present two algorithms which will use the concept of a divide and conquer algorithm. Sentences similar to our example here (with connectors between parts of a sentence) will be split up to take advantage of the divide and conquer concept. When we do this, we have to keep in mind that we use Horn's Theorem.

1.2.2 Psychological Predicate contra logical Predicate

Another thing that has to be noticed is that it seems to be normal to take the grammatical predicate as the logical one.
E.g.: If we have the simple sentence:

Socrates is mortal

We would translate it in the common way as

$$mortal(Socrates)$$

but we could also translate in the following way

$$Socrates(mortal)$$

Where the subject becomes our main logical predicate, which in some cases may express the psychological intentions of the speaker better than the classical one.
But if we want to transform sentences a bit more complicated it is more useful to remain at the classical interpretation to avoid getting in trouble with the formalism.

Chapter 2

Developing the Algorithms

Now we want to develop algorithms that fulfil our desires. Later we will implement our algorithms in Prolog, so that we will try to take advantage of some Prolog-features when developing it. Of course the first thing we need is a good parser to get the syntactic structure which will be the base of the translations.

2.1 The Parsing

The word parsing is derived from the Latin phrase pars orationis, or "part of speech", and refers to the process of assigning a part of speech (noun, verb, and so on) to each word in a sentence and grouping the words into phrases. One way of displaying the result of a syntatctic analysis is with a parse tree. A parse tree is a tree in which interior nodes represent phrases, links represent applications of grammar rules and leaf nodes represent words. We define the yield of a node as the list of all leaves below the node, in left-to-right order, then we can say that the meaning of a parse tree is that each node with label X asserts that the yield of the node is a phrase of category X.
We will use such parse trees to represent grammatic structures, since they are very easy to read and understand.
There are many algorithms for parsing - recovering the phrase structure of an utterance, given a grammar. We will use the topdown algorithm implemented in Prolog and do not bother more about this, although there may be faster and better ones.
Our grammar will be based on the idea of phrase structure (that strings are composed of substrings called phrases, that come in different categories). It will be a context free grammar (CFG). Although it is widely accepted that at least a few natural languages are not context-free (see [5] for details),

English seems to be context-free Normally CFGs are represented in Backus-Naur-form (BNF). We will use the Prolog-feature to present our grammar in Definite Clause Grammar (DCG) formalism in the implementation of our algorithms. Every good Prolog-system has the built-in feature to transform DCG-rules automatically in Prolog-rules.
But before we start to define our grammar we have to be aware that we keep it small. Therefore we need

2.1.1 The Dictionaries

If we want to be as general as possible we cannot include rules for every possible word in our grammar. Since English consists of about 400.000 words, this would extend the possibilities of our parser due to the extraordinary amount of possible combinations that would have to be checked. So it is useful to use one or more dictionaries which whom we can find out the part of speech of the in the words in the utterance. According to the search result we create rules which we add to our grammar. We use two dictionaries. They are called `moby.lex` and `link.lex` which have been created by changing slightly `mpos.tar.Z` and `words` which can be found at `http://www.dcs.shef.ac.uk/research/ilash/Moby/mpos.html` and `http://bobo.link.cs.cmu.edu/grammar/html/ftp_site/link-grammar/system4.1/link4.1/data/words`. The changes were necessary to create an equal format for the search algorithm. The dictionaries are ordered alphabetically and after every word separated by a field delimiter of (ASCII 215) can be found a list of part of speech symbols. This symbols are:

Part of Speech	Symbol
noun	N p h L m
verb	V i t l q
modal verb	M
adjective	A a
article	D I d
adverb	v e
conjunction	C c
preposition	P
pronoun	r
gerund	G g

Now we can take a closer look at our grammar.

2.1.2 The Grammar

Of course our grammar is not a full English grammar. It will only cover a part of the enormous variety of English sentences and the language defined by it will only be a subset of English. Of course we are only interested in statements and do not cover questions or orders. It may also be possible to create sentences that do not occur in natural English, but since we expect to get only syntactically correct inputs we do not bother about this.
As said we use a CFG that we will call G. We define it as the quadruple (V, W, P, S) where:

- V is the set of nonterminal symbols or variables. They are explained in a table below

- W is the set of terminal symbols, which are simply all words used in English
- P is the set of productions.
- S is the starting variable $\in V$. It is `sentence`.

The productions P can be found on appendix A on page 59. We will call the language defined by G LOGENG (short for Logical English). To express it mathematically:

Definition 2.1.1 $LOGENG = L(G)$

The following table will give a short explanation of the occurring variables V in the grammar. For further interest please check the grammar itself.

Variable	Description	Example
neg	negation of verbs	not
conj1	simple connector	and, or, ","
conj2	advanced connector	although
conj3	advanced connector	but
cond1	conditional connector placed at beginning or in between of sentences	if, as
cond2	conditional connector placed only in between of sentences	therefore, so
rpn1	relative pronoun	who, which
rpn2	relative possessive pronoun	whose
quantA	universal quantifier	every, all
quantE	existential quantifier	a, some
art	article	the
adj	adjective	old, strong
adv	adverb	yesterday, often, lovely
noun	noun	man, hill
pron	pronoun	me, you, he
prep	preposition	on, of, from
propname	proper name	Franz, Socrates
vb	verbform of be	am, was ,are
mv	modal verb	can, may, must
verb	all other verbs	eat, love
gerund	gerund	eating, loving
verbgroup	predicate of a sentence	has always thought and done
verbgroupend	predicate without the auxiliary verbs	never gone
verbphrase	sentence without the subject	buys her flowers
adjectivphrase	list of adjectives	young, strong and healthy
nounphrase	subject or object	the young man
prepphrase	attribute with preposition	on the hill
adverbphrase	list of adverbs of time matter and place	yesterday in Austria
objectphrase	objects of a sentence	mum and dad flowers
object	one single object	flowers
relativeclause	relative clause	who called me

Every other variable stands for itself and has the same name as a terminal symbol. Like `have` is a variable for "have" and so on.
When we use the grammar in DCG-format to parse a sentence in Prolog we get a list of lists in which the parsing data is stored. Since this list is hierarchically it is very useful for printing the parsing tree or for using a divide and conquer algorithm. Wherever it seems useful in this work we will use the parsing tree representation as explained above for complex lists instead of the lists itself to create a better imagination of the parsing data. The algorithm used for this can be found and is explained in [3].
In this lists (and in the parsing trees) we use shortcuts for some of the variables described above. We also use strings in capital letters to represent The following table shows the shortcuts.

Variable	Shortcut
`verbgroup`	vg
`verbgroupend`	vge
`verbphrase`	vp
`adjectivphrase`	ap
`nounphrase`	np
`prepphrase`	pp
`adverbphrase`	advp
`objectphrase`	op
`object`	obj
`relativeclause`	rc
`sentence`	s

All other variables do not have shortcuts. From now on we will use this lists (and when useful parsing trees) to represent our sentences and phrases instead of their grammatical rules.

2.2 The Transformation

After the parsing we have all information we need to transform the sentence in an "equivalent" string in Propositional or First-order Logic. Of course this equivalence is not a certain one. We will see more about this in chapter 3 on page 40. The most important information is the position of the connectors, negations and quantifiers as well as their scope.
We will start with the transformation into Propositional Logic, since we can take advantage of many of the things used in this algorithm for the transformation into First-order Logic (like the treatment of connectors).

2.3 Transformation into Propositional Logic

2.3.1 Principal Things about Connectors

First of all we have to think about the translation of linguistic connectors into logical ones. This problem is quite difficult and ambiguous. Also the number of possible connectors is quite big, and we only try to find a transformation for the connectors included in our grammar (see appendix A on page 59). It is easy to extend the grammar by new connectors and extend the algorithm by the rule for his transformation.
The connectors and transformations we will use can be seen below.

and: This connector is easy to transform. The linguistic and logical semantics are nearly the same. It is obvious that we represent it with $\wedge$.

or: This is a typical example for an connector whose logical and syntactical semantics sometimes are different. E.g.:

John eats fish or Susan eats steak.

is equivalent to

(John eats fish) $\vee$ (Susan eats steak)

Whereas in

You can come with me or stay here.

the "or" represents something like "either ... or" ($\oplus$ in logic). Without understanding something of the context of the phrase its impossible to differ between this two significations. We will represent "or" with $\vee$ since this seems more natural.

",": Can be either "or" or "and". Normally it should be transformed like "and" but in case of an list with "or" like in

You can come on Monday, Tuesday or Wednesday.

it has to be transformed like "or". We have to search the result of the parsing if there is an "or" at he same level of conjunction and decide then how we translate it. There also might be other possibilities where it should be transformed into "$\rightarrow$". See below:

as: There are two different possibilities how this connector can occur in a sentence. (There may be more but we only look at this two.)

1. It is followed by two sentences separated by a comma. E.g.:

   As I'm working, I earn money.

   It is obvious that we transform the ',' into "$\rightarrow$" and remove the "as". So we get:

   (I'm working) $\rightarrow$ (I earn money)

2. It stands between two subsentences. E.g.:

   I earn money as I'm working.

   It is obvious that we transform the connector ("as" in this case) into "$\leftarrow$" and so we get:

   (I earn money) $\leftarrow$ (I'm working)

because: Is equivalent to "as".

since: Is equivalent to "as".

if: Is equivalent to "as" but we have to remind that it can occur that we have a "then" instead of the comma. Then we have to transform this "then" into the implication symbol.

therefore: Is equivalent to "as", but only the 2^{nd} case occurs.

thus: Is equivalent to "as", but only the 2^{nd} case occurs.

so: Is equivalent to "as", but only the 2^{nd} case occurs.

The following connectors do not have something like a natural transformation, but we will substitute them by "and" and add an additional statement.

although: In this case we add "$\wedge$ (Event is unexpected)". E.g.:

I am drinking wine although I love beer.

becomes to

(I am drinking wine) $\wedge$ (I love beer) $\wedge$ (Event is unexpected)

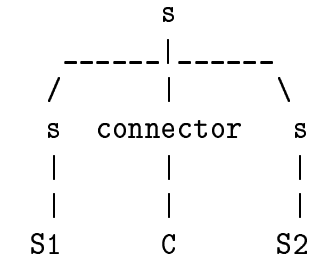
but: we add "$\wedge$ (Events are contrary)".

however: here we add "$\wedge$ ((Events are contrary) $\vee$ (Event is unexpected))".

2.3.2 The Algorithm TPROP

We call or algorithm TPROP (for Transformation into PROPositional logic). It will be a divide and conquer algorithm. So that it will happen various times that we will restart TPROP with a subsentence of our input.
We will see now the main steps of TPROP.

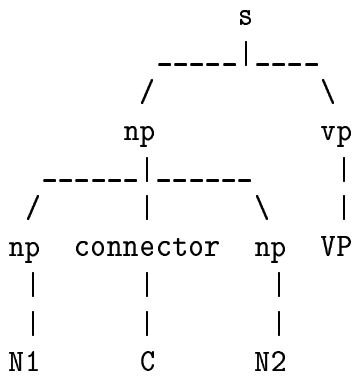
1. If the sentence consists of more than one sentence separated by a connector, - that means it can be represented by



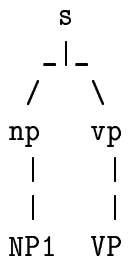
   - we replace this connector according to the rules in section 2.3.1 on page 12 and restart TPROP with every one of this sentences ($s(S1)$ and $s(S2)$).

2. Now we have to look if we still have connectors within the sentence. If so, we replace it like above and create two sentences out of it by separating the part including the connector and combining it with the rest of the sentence. Like in step 1 we restart TPROP with every new sentence. To avoid problems of contradictory interpretations we have to keep a certain order in searching for connectors in sentence parts. The following list shows the priorities and the rules how to create the new sentences.

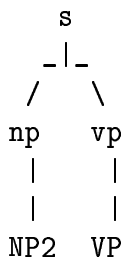
   (a) **subject:** If our sentence is of the form



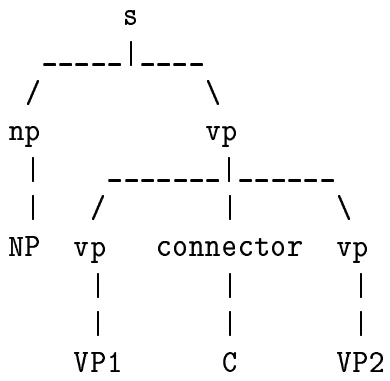
we get the two new sentences



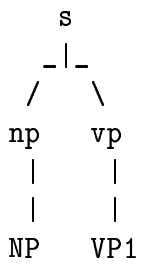
and



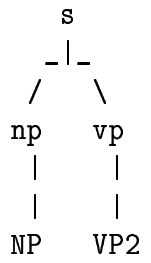
(b) **verbphrase:** If our sentence is of the form



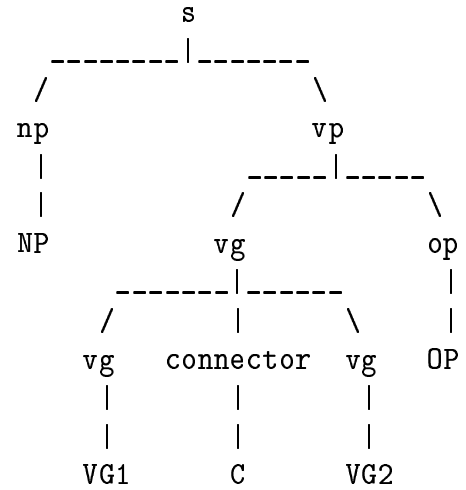
we get the two new sentences



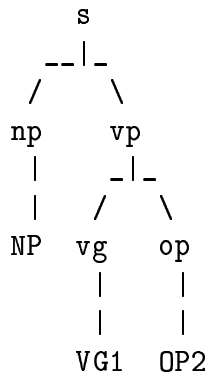
and



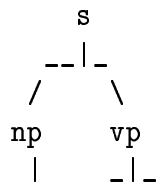
(c) **verbs:** If our sentence is of the form

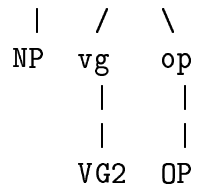


we get the two new sentences

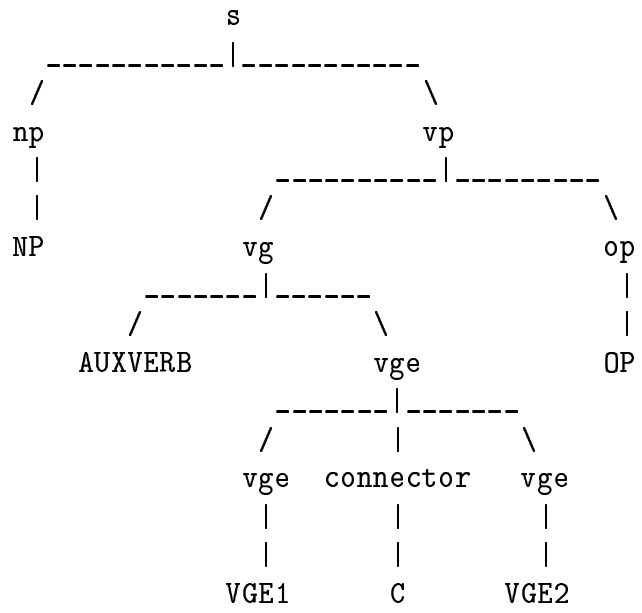


and

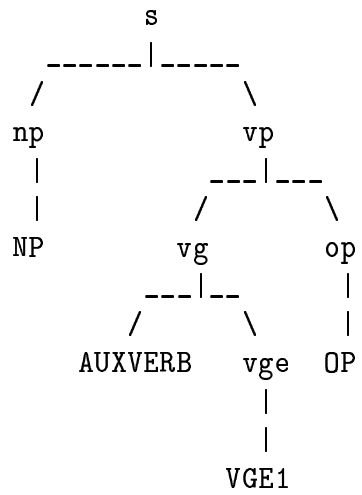




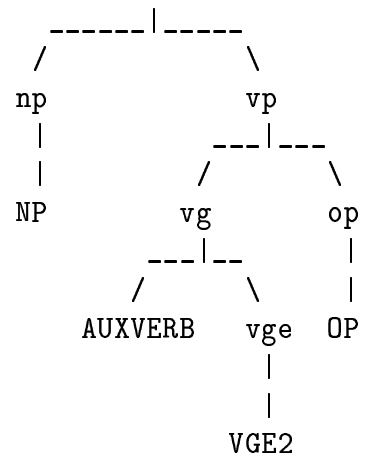
(d) **verbparts:** If our sentence is of the form



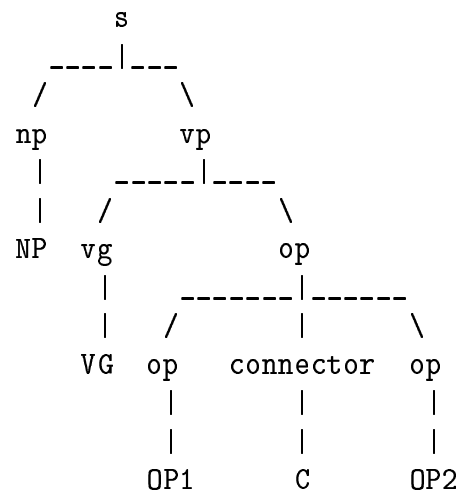
we get the two new sentences



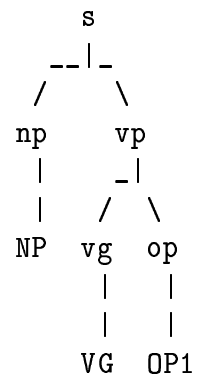
and



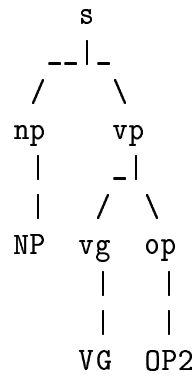
(e) **objectphrase:** If our sentence is of the form



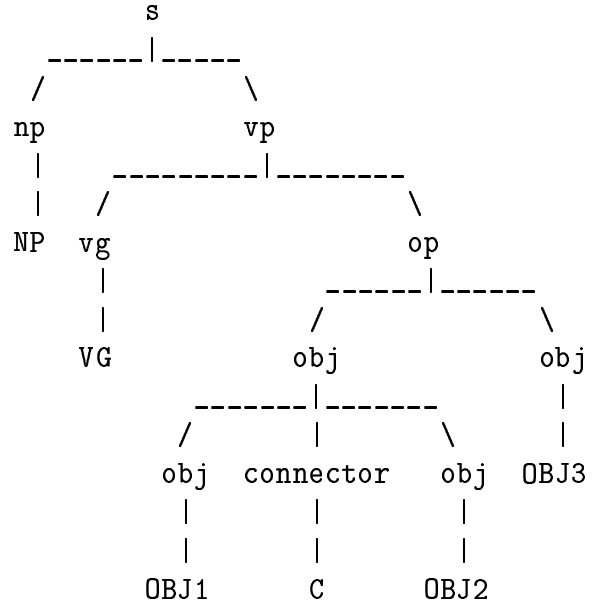
we get the two new sentences



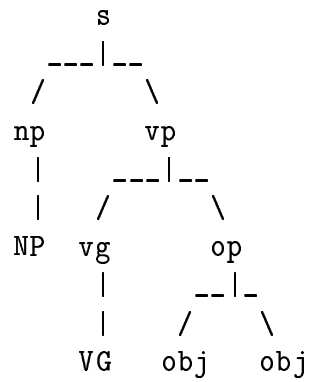
and

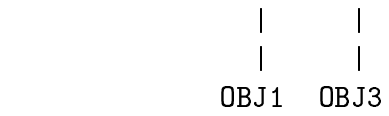


(f) **first object:** If our sentence is of the form

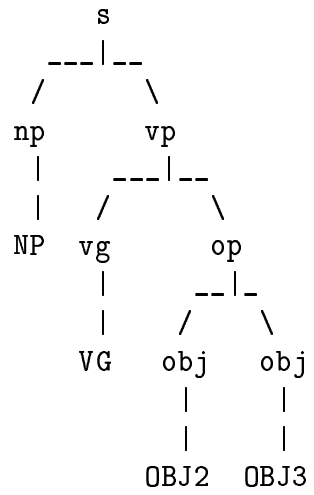


we get the two new sentences (there may be no third object, but this can be solved analogue).

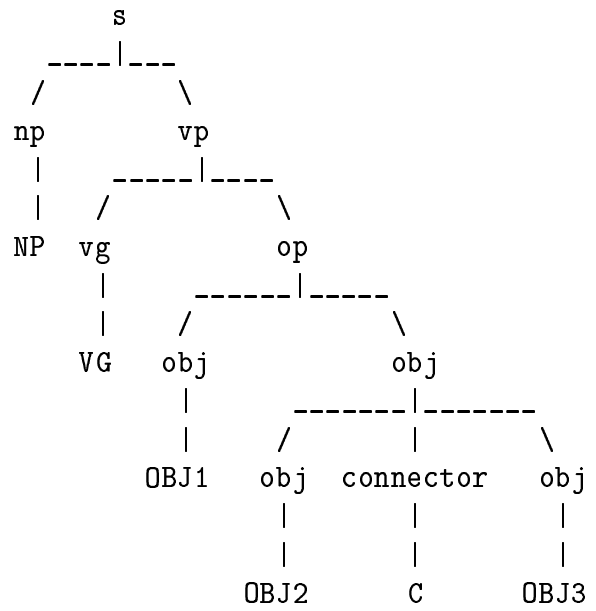




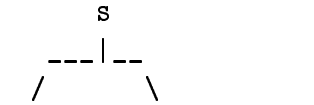
and

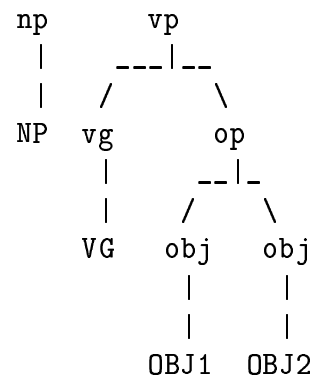


(g) **second object:** If our sentence is of the form

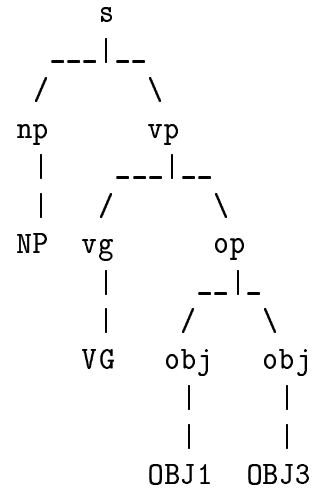


we get the two new sentences





and



With every of our new sentences we restart TPROP instead of continuing the algorithm.

3. Finally look for negations in the phrase and make them to the $\neg$ operator. There are two possibilities for our negation. According to this we have to act in the following way:

   no replace it by "a/some" and put $\neg$ before the sentence

   not just delete it and put $\neg$ before the sentence.

2.3.3 Examples

To illustrate the above algorithm we look at the following examples:

Example 1

We want to translate the following sentence:

> If the sun shines, it does not rain and if it rains, the sun does not shine.

We start with step 1 and since our sentence consists of two sentences connected by the connector "and" we get:

> (If the sun shines, it does not rain) $\wedge$ (if it rains, the sun does not shine.)

With every of our new sentences we restart TPROP. Both consist of another two sentences combined by "if". We get:

> ((the sun shines) $\rightarrow$ (it does not rain)) $\wedge$ ((it rains) $\rightarrow$ (the sun does not shine)).

No more sentence can be split up so the next time restarting with our algorithm we move on to step 2. But as we can see this too won't change anything in this case. We move on to step 3, were we move the "not's". We get:

> ((the sun shines) $\rightarrow \neg$ (it does rain)) $\wedge$ ((it rains) $\rightarrow \neg$ (the sun does shine)).

The example above shows that we may have problems with semantically equivalent expressions like "does shine" and "shines". More about this in section 3.2 on page 44.

Example 2

With this example we would like to demonstrate step 2 of our algorithm.

> Susan and Ann will drink coffee or eat chocolate.

Step 1 does not bring any change so we move on to step 2, where we first look in the subject to find connectors. We get:

> (Susan will drink coffee or eat chocolate) $\wedge$ (Ann will drink coffee or eat chocolate)

Applying step 2 once again with every subsentence we detect the connector between verbparts. We separate and get:

((Susan will drink coffee) $\vee$ (Ann will to drink coffee)) $\wedge$ ((Ann will eat chocolate) $\vee$(Ann will eat chocolate))

Since we have no negations in or sentences step 3 does not change anything. And we already have or final result. It is important to keep the order given by step 2, to avoid confounding results(in our case the parentheses would change).

2.4 Transformation into First-order Logic

2.4.1 Principal things about quantifiers

Like the connectors the quantifiers in natural logic are not that easy to translate. There are cases where we would need semantic knowledge to translate it. E.g.:

A man needs a beer.

can either mean that every man needs a beer or that one certain man needs a beer. So the possible transformations into predicate logic would be:

$$\forall x_1(man(x_1) \rightarrow (\exists x_2(beer(x_2) \wedge needs(x_1, x_2))))$$

or

$$\exists x_1(man(x_1) \wedge \exists x_2(beer(x_2) \wedge needs(x_1, x_2)))$$

The same problem can occur if we look at the meaning of the linguistic quantifier "one" which in some cases also can stand for everyone. E.g.: "One must learn to keep quiet". And of course there can be found a huge number of examples like this. Also sometimes there is no quantifier and without knowing the semantic background we simply cannot say if this means "all" or "some". E.g. in: "Children go to school". So how do we translate all this? The best way would be to transform it into a $\exists$ wherever any doubt exist of the right transformation. If we would translate it into $\forall$ we might generate a generalisation which was not meant by the speaker. Whereas in the other case the translation with $\exists$ still keeps being certain, although the speaker meant $\forall$.
So we transform the quantifiers occurring in the grammar in the following way:

"every", "all" and "each" into $\forall$
"some","one","a", "an" into $\exists$
and if there is no quantifier we suppose it is meant $\exists$

2.4.2 The Algorithm TPRED

We call or algorithm TPRED (for Transformation into logic of PREDicates). Before we start to parse the sentence it is useful to replace some quantifiers to avoid problems with the separation of quantifier and predicate. Table 2.1 shows the replacements we will use (of course there may be others).

original	replaced by
nobody	not somebody
nothing	not something
everybody	every person
everyone	every person
everything	every thing
anybody	every person
anyone	every person
anything	every thing
someone	some person
somebody	some person
something	some thing
any	every

Table 2.1: Replacements for First-order Logic

After having parsed the sentence we have, as in the algorithm for the Propositional logic, all information we need to make our transformation. We will give some of the steps below names to make this implementation better readable. We can imagine them as the names of the procedures representing them in an implementation
We will also use some functions, which will be explained now:

Definition 2.4.1 *$\mathcal{Y}$ is a function mapping the nodes of parsing tree to a string, Where $\mathcal{Y}(X)$ is the yield (phrase) of the node X with "_" instead of " " to separate the words.*

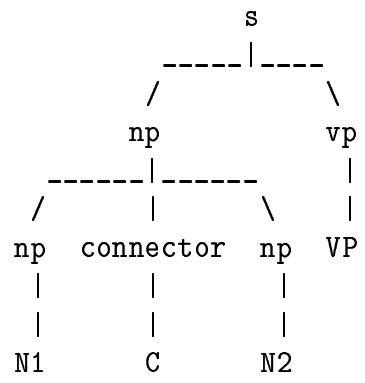
Definition 2.4.2 *$\mathcal{T}$ is a function from LOGLAN into the set of logic connectors. It maps linguistic connectors into logic connectors according to 2.3.1 on page 12.*

With the above definitions we will now describe our algorithm TPRED.

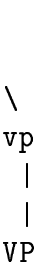
Description of TPRED:

1. The first step is the same as in TPROP. We search for connectors separating sentences and restart TPRED with its subsentences. (Like TPRED, TPROP is a divide and conquer algorithm)

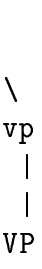
2. **sepSub:** Now we cannot do the same like in the propositional algorithm and generate new sentences if there are connectors in between parts of our sentence. We can do this only if the subject of our sentence consists of more than one subsubject. That means: If our sentence is of the form:



we restart sepSub with



and



But before we do that we have to look if our sentence contains any adverb phrases representing adverbs of time manner and place. We will add them to the vg-part of VP and use it later.

3. **transNP:** Now we transform the subject into First-order Logic. We have to store two types of information. First we save the name of the variable or constant representing our subject and second we have to take care that we do not use the same variable to represent two different things. (We name them $x_1, x_2, \ldots$ and save the last number used).

   (a) If the subject consists of a proper name or a personal pronoun (It has the format ($np(propname(X))$ or $np(pron(X))$, we use this X as a constant, exit transNP and go on to step 4 (sepVP). If not, we use a new variable to represent it and go to the next step.

   (b) Now we have to look if there is an universal quantifier used in our subject (it's of the form $np(quantA(Q), X)$ where Q is the quantifier) (see section 2.4.1 on page 23 for more information).

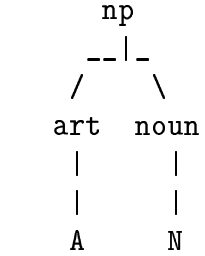
   If so, we start with "$\forall x_n($" and terminate with "$\rightarrow$ (" else if the quantifier is "no" (the subject is of the format $np(no, X)$), we start with "$\forall x_n($" and terminate with "$\rightarrow \neg($".

   If neither of this two cases is true we start with "$\exists x_n($", terminate with "$\wedge$" and remove (if the subject is of the form $np(quantE(Q), X)$) an eventually existing existential quantifier. The missing middle part is created out of X in the next substep.

   (c) The rest of or subject (represented by X in the last step) has to be transformed into predicates with the subject representing constant or variable as argument. We will call this transformation **TraSubPred** since the definition of this step will be recursive we will need this in its definition.

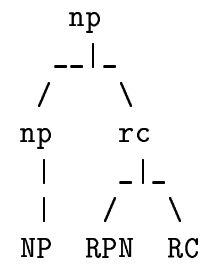
   The following cases (or a combination of them) have to e treated by TraSubPred:

   - *the kernel of subject:*
     If the subject is of the form



or of form np(N), it is transformed into a $A\_N(x_n)$ or $N(x_n)$ (e.g.: "the knife"" is transformed into $the\_knife(x_n)$)

- *Relative clauses referring to the subject:*
  If of form



  Then we restart TraSubPred with $np(NP)$ and combine the result with the transformation of

$$rc(RPN, RC)$$

  which becomes either to

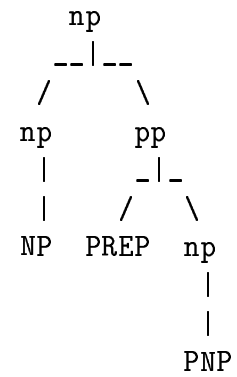
$$\wedge \quad \mathcal{Y}(RC)(x_n)$$

  if $RPN \neq rpn1(rpn1(who))$
  or else

$$\wedge \quad \mathcal{Y}(RPN)\_\mathcal{Y}(RC)(x_n)$$

  (e.g.: "a women who works" is transformed into $a\_women(x_n) \wedge works(x_n)$)

- *Prepositional phrases referring to the subject:*
  If of form



  Then again we restart TraSubPred with $np(NP)$ and combine the result with the transformation of

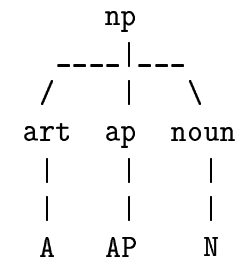
$$pp(PREP, np(PNP))$$

which becomes to

$$\wedge \quad \mathcal{Y}(PREP)\_\mathcal{Y}(PNP)(x_n)$$

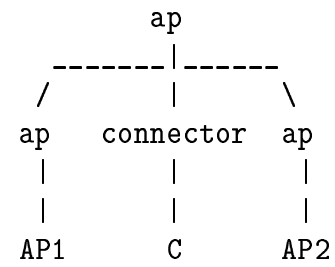
(e.g.: "the man from Italy" is transformed into $the\_man(x_n) \wedge from\_Italy(x_n)$)

- *Adjective attributes:*
  If of form



again we restart TraSubPred with $np(art(A), noun(N))$ combine the result with an "$\wedge$" and the transformation of $ap(AP)$ which is transformed in a divide and conquer algorithm. We call it ADTRAN:

If $ap(AP)$ is of the form



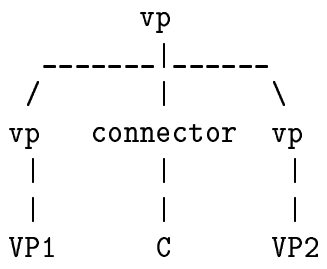
We restart ADTRAN with $ap(AP1)$ and $ap(AP2)$. The results are separated by $\mathbf{T}(C)$ and surrounded by parenthesis.
Else $ap(AP)$ has the form $ap(adj(adj(ADJ)))$ where ADJ is a single adjective attribute. we transform it into $ADJ(x_n)$ and return this.

(e.g.: "this young man" is transformed into $this\_man(x_n) \wedge young(x_n)$)

To illustrate a combination of the above cases we take a look at "the young or silent man from Italy" which is transformed into $the\_man(x_n) \wedge (from\_Italy(x_n) \wedge (young(x_n) \vee silent(x_n)))$

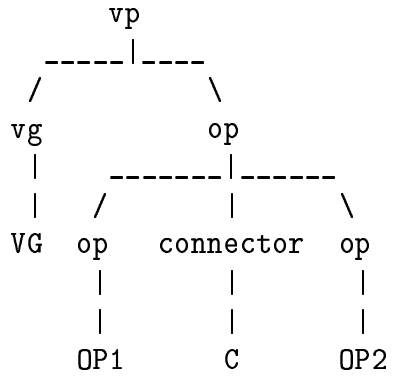
(d) Before we go to the next step we have to save the number of parentheses that remain open, we will need this information in one of the last steps.

4. **sepVP:** Once having transformed the subject we look at the remaining verbphrase. If it has the form:

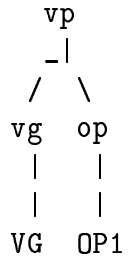


   we transform the connector C into a logical connector and restart sepVP with $vp(VP1)$ and $vp(VP2)$ (again we divide and conquer the problem). After having transformed them we return it in the following way $(sepVP(VP1) \quad \mathbf{T}(C) \quad sepVP(VP2))$.

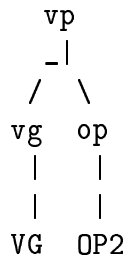
5. **sepOP:** Now we look at the objectphrase of our verbphrase. If the verbphrase has the form



   than we restart sepOP with

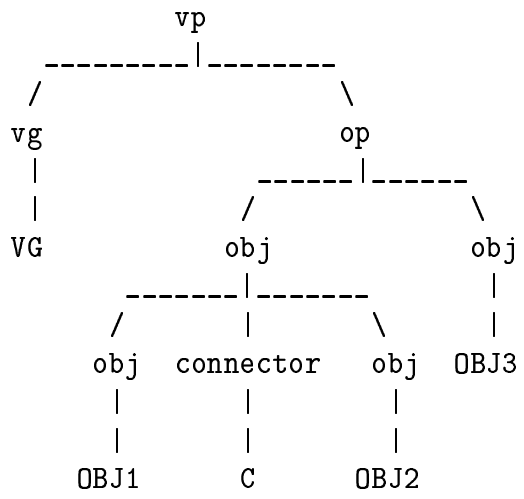


   and

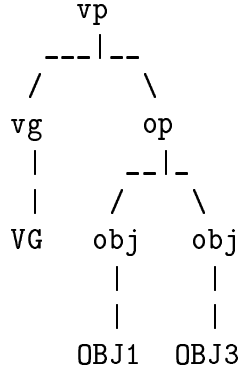


and return the results separated by $\mathcal{T}(C)$ and surrounded by a parenthesis.

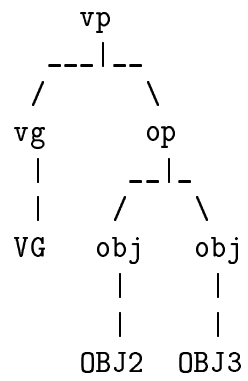
6. **sepOBJ:** We do the same as before but now with the object. If there are two objects, just with the first one. Again we look at the verbphrase. If our verbphrase is of the form:



we restart sepOBJ with



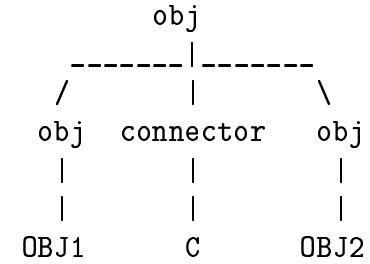
and



If object represented by $OBJ3$ does not occur we act analogue. Again the results are separated by $\mathcal{T}(C)$ and surrounded by parenthesis.

7. Now we transform the first object in its predicate form. We do this in exactly the same way as in step 3 with the subject by using transNP to.

8. **sepOBJ2:** What is left to separate now is the second object (if it exists and has the form:

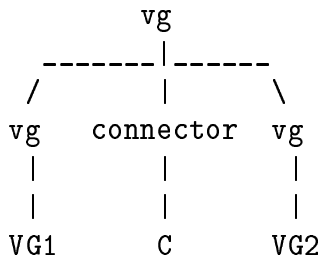


). Now we do not combine it with the rest of our verbphrase but we remember the verbpart represented in the above step by $vg(VG)$ for further use. Of course we have to repeat sepOBJ2 with $obj(OBJ1)$ and $obj(OBJ2)$ and separate their results by $\mathcal{T}(C)$ and surround them by parenthesis.

9. Now we transform the second object (if it exists) in its predicate form. Once again we do this in exactly the same way as in step 3 with the subject by using the same procedure transNP.

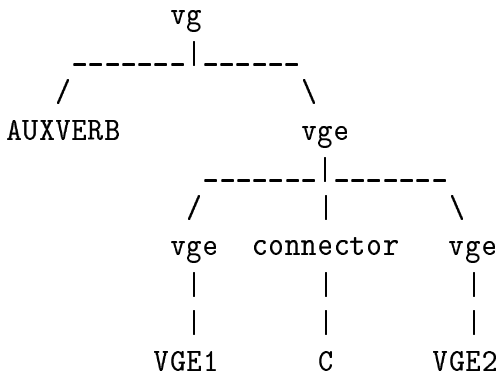
10. **sepVG:** Before we start to create the predicates, we have to take a look at our verbpart , which is represented by $vg(VG)$. There are two cases where we have to do something:

    - It has the form

      Then we restart sepVG with $vg(VG1)$ and $vg(VG2)$ and after having transformed them completely we separate them by $\mathcal{T}(C)$.

    - or it has the form:

      Then we restart sepVG with $vg(AUXVERB, vg(VGE1))$ and $vg(AUXVERB, vge(VGE2))$ and again after having transformed them completely we separate them by $\mathcal{T}(C)$.

11. **TransVG:** After that we transform the verbpart VG into a predicate, having one, two or three arguments depending at the number of objects. As the verbpart includes also adverbs of time, matter an place this adverbs are integrated in the predicate. We use the $\mathcal{Y}$ function to transform them.

12. **ClosePar:** Finally we close all opened parenthesis (as we have stored them in a list too, this does not cause any problem).

13. **Post:** What is left now is some postprocessing. We change the predicates $is(\ldots)$ and $are(\ldots)$ into $equal(\ldots)$ Finally, every subformula of the form $\exists x_i P(x_i) \wedge equal(a, x_i)$ is transformed into $P(a)$.

We will now write TPRED in pseudocode and use the names of the procedures to represent the steps above. (if a step does not have a procedure name, it is included in the anterior procedure) The conditions of the if and case statements are written very sloppy. The exact conditions and explanations can be seen in the text version above.

TPRED in pseudocode:

```
global variable:
varNum ← 1

procedure TPRED(textS)
changedS ← change(textS)
S ← parse(changedS)
if S = S1,C,S2 then
    return "(" + TPRED(S1) +T(C)+ TPRED(S2) + ")"
else return Post(sepSub(S))

procedure sepSub(S)
save Adverbs of time, manner or place in the Vg-part of S
NP,VP ← S
if NP = NP1,C,NP2 then
    begin
        S1 ← NP1,VP
        S2 ← NP2,VP
        return "(" + sepSub(S1) +T(C)+ sepSub(S2) + ")"
    end
else return transNP(NP,var,par) + sepVP(VP,var) + ClosePar(par)

procedure transNP(NP,var,par)
case NP = PROPNAME
    begin
        var ← PROPNAME
        par ← 0
        return <empty string>
    end
case NP = PRON
    begin
        var ← PRON
        par ← 0
```

```
            return <empty string>
        end
case NP = QuantA,X
        begin
            var ← "x" + varNum
            varNum ← varNum +1
            par ← 2
            return "∀" + var + "(" + TraSubPred(X,var) + "→ ("
        end
case NP = NEG,X
        begin
            var ← "x" + varNum
            varNum ← varNum +1
            par ← 2
            return "∀" + var + "(" + TraSubPred(X,var) + "→ ¬("
        end
case (NP = QuantE,X) or (NP = X)
        begin
            var ← "x" + varNum
            varNum ← varNum +1
            par ← 1
            return "∃"+var+"(" +TraSubPred(X,var)+"∧"
        end

procedure TraSubPred(NP,var)
case (NP = ART,NOUN) or (NP = NOUN)
        return 𝒴(NP) + "(" + var + ")"
case NP = NP1,RC
        if RC = who,RC1 then
            return TraSubPred(NP1,var)+"∧"+𝒴(RC1)+"("+var+")"
        else
            return TraSubPred(NP1,var)+"∧"+𝒴(RC)+"("+var+")"
case NP = NP1,PP
        return TraSubPred(NP1,var)+"∧"+𝒴(PP)+"("+var+")"
case NP = ART,AP,NOUN)
        begin
            NP1 ← ART,NOUN
            return TraSubPred(NP1,var) + "∧" + ADTRAN(AP,var)
        end
case NP = AP,NOUN)
        return TraSubPred(NOUN,var) + "∧" + ADTRAN(AP,var)
```

```
procedure ADTRAN(AP,var)
if AP = AP1,C,AP2 then
    return "("+ADTRAN(AP1,var)+𝒯(C)+ ADTRAN(AP2,var)+")"
else
    begin
        ADJ ← AP
        return 𝒴(ADJ) + "(" + var + ")"
    end

procedure sepVP(VP,var)
if VP = VP1,C,VP2 then
    return "("+sepVP(VP1,var)+𝒯(C)+sepVP(VP2,var)+")"
else return sepOP(VP,var)

procedure sepOP(VP,var)
VG,OP ← VP if OP = OP1,C,OP2 then
    begin
        VP1 ← VG,OP1
        VP2 ← VG,OP2
        return "("+sepOP(VP1,var)+𝒯(C)+sepOP(VP2,var)+")"
    end
else return sepOBJ(VP,var)

procedure sepOBJ(VP,var)
VG,OP ← VP
case OP = OBJ,OBJ2
    if OBJ = OBJ3,C,OBJ4 then
        begin
            VP1 ← VG,OBJ3,OBJ2
            VP2 ← VG,OBJ4,OBJ2
            return "("+sepOBJ(VP1,var)+𝒯(C)+sepOBJ(VP2,var)+")"
        end
    else
        begin
            OBJre ← transNP(OBJ,var2,par)
            vars ← var + "," + var2
            return OBJre+sepOBJ2(OBJ2,VG,vars)+ClosePar(par)
        end
case OP = OBJ1
    if OBJ1 = OBJ3,C,OBJ4 in VP then
```

```
            begin
                VP1 ← VG,OBJ3
                VP2 ← VG,OPJ4
                return "("+sepOBJ(VP1)+T(C)+sepOBJ(VP2)+")"
            end
        else
            begin
                OBJre ← transNP(OBJ1,var2,par)
                vars ← var + "," + var2
                return OBJre+sepVG(VG,vars)+ClosePar(par)
            end
case OP = <empty string>
        return sepVG(VG,var)

procedure sepOBJ2(OBJ,VG,vars)
if OBJ = OBJ1,C,OBJ2 then
        return "("+sepOBJ2(OBJ1,VG,vars)+T(C)+sepOBJ2(OBJ2,VG,vars)+")"
else
        begin
            OBJre ← transNP(OBJ,var3,par)
            vars ← vars + "," + var3
            return OBJre+sepVG(VG,vars)+ClosePar(par)
        end

procedure sepVG(VG,vars)
if VG = VG1,C,VG2 then
        return "("+sepVG(VG1,vars)+T(C)+sepVG(VG2,vars)+")"
else if VG = AUXVERB,VGE1,C,VGE2 then
        begin
            VG1 ← AUXVERB,VGE1
            VG2 ← AUXVERB,VGE2
        return "("+sepVG(VG1,vars)+T(C)+sepVG(VG2,vars)+")"
        end
else return TransVG(VG,vars)

procedure TransVG(VG,vars)
return Y(VG) + "(" + vars +")"

procedure Post(S)
for SubS = Substring of S do
        if (SubS = is(X)) or (SubS = are(X)) then
```

SubS $\leftarrow equal(X)$
for SubS = Substring of S **do**
if SubS $= \exists x_i P(x_i) \wedge equal(a, x_i)$ **then**
SubS $\leftarrow P(a)$
return S

2.4.3 Examples

To illustrate TPRED we look at the following examples. We transform them parallel to get a better imagination of the algorithm.

Example 1

We want to translate the following sentences:

> If every grandmother is a mother and Anne is a grandmother, Anne is a mother.

Applying step 1 we get:

> (every grandmother is a mother and Anne is a grandmother) $\rightarrow$ (Anne is a mother).

Once again step 1:

> ((every grandmother is a mother) $\wedge$ (Anne is a grandmother)) $\rightarrow$ (Anne is a mother).

Now step one is finished so we move to step 2. But since we only have simple subjects we move on to step 3. Here we choose x_1 as a representation of grandmother and since we have the quantifier "every" the subject in the first sentence is transformed into $\forall x_1(grandmother(x_1) \rightarrow$ (. "Anne" is a constant and will be stored.
The steps 4,5 and 6 do not have effect for this example so we move on to step 7, where we transform the object into a predicate form. In all three cases we have an existential quantifier. So the result of the transforming can be shown in the following table.

a mother	$\exists x_2(mother(x_2) \wedge$
a grandmother	$\exists x_3(grandmother(x_3) \wedge$
a mother	$\exists x_4(mother(x_4) \wedge$

Step 8, 9 and 10 have once again no effect so that we move on to step 11, where we transform the verbparts into their predicate form. In all three cases our verb is "is", which is transformed into the predicate $is(\ldots)$ with two arguments. After closing the parenthesis in step 12 we get:

$$\begin{array}{c}((\forall x_1(grandmother(x_1) \rightarrow (\exists x_2(mother(x_2) \wedge is(x_1, x2)))))\\ \wedge(\exists x_3(grandmother(x_3) \wedge is(Anne, x3))))\\ \rightarrow (\exists x_4(mother(x_4) \wedge is(Anne, x4)))\end{array}$$

And finally after applying step 13 we have:

$$\begin{array}{c}((\forall x_1(grandmother(x_1) \rightarrow (\exists x_2(mother(x_2) \wedge equal(x_1, x2)))))\\ \wedge grandmother(Anne)) \rightarrow mother(Anne)\end{array}$$

Example 2

Now we want to translate this:

> On Wednesday everyone buys his mum flowers or a box of chocolate

First we transform "everyone" into "every person" and since our sentence does not consist of more than one subsentence we move on to step 2 where we also find only one subsubject that we transform in step 3 into:

$$\forall x_1(person(x_1) \rightarrow ($$

In step 4 we have nothing to do, as we have only one subverbphrase. And in step 5 and 6 its the same. We neither have a subobjectphrase nor a subobject of the first object.
In step 7 we transform the first object "his mum" according to the rules in step 2 into

$$\exists x_2(his\_mum(x_2) \wedge$$

Now we have to apply step 8 since the second object consists of 2 subobjects. They are combined by "or" which is transformed into $\vee$. We remember the verbpart, which is the adverb of time "On Wednesday" combined with the verb "buys" and apply step 9, which is equivalent to step 2 with "flowers" and "a box of chocolate". From the first subobject we get:

$$\exists x_3(flowers(x_3) \wedge$$

The second subobject is a bit more complicated as we have to transform an prepositional phrase too. According to the rules of step 3c we get:

$$\exists x_4(box(x_4) \wedge of\_chocolate(x_4) \wedge$$

And move onward to step 11 where we transform the verbpart. We have three arguments, since we have two objects. And for every subobject we have to apply this transformation and unite it with the result of the object transformation. So we get:

$$\exists x_3(flowers(x_3) \wedge on\_Wednesday\_buys(x_1, x_2, x_3))$$

$$\exists x_4(box(x_4) \wedge of\_chocolate(x_4) \wedge on\_Wednesday\_buys(x_1, x_2, x_4))$$

Finally we combine it with the transformation of the connector and the rest of our transformation and close the opened parenthesis. We get:

$$\begin{gathered}\forall x_1(person(x_1) \rightarrow (\exists x_2(his\_mum(x_2) \wedge \\ (\exists x_3(flowers(x_3) \wedge on\_Wednesday\_buys(x_1, x_2, x_3)) \vee \\ \exists x_4(box(x_4) \wedge of\_chocolate(x_4) \wedge on\_Wednesday\_buys(x_1, x_2, x_4))))))\end{gathered}$$

and are ready since the postprocessing of step 13 does not change anything.

Chapter 3

The Limits

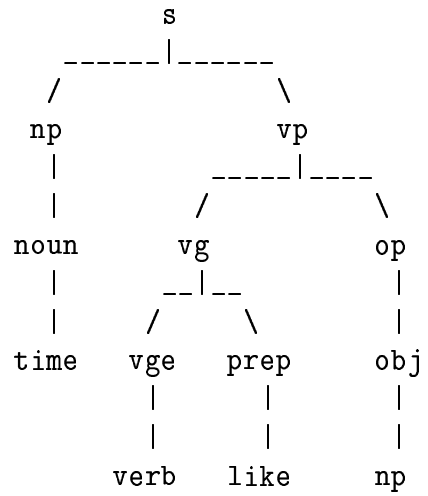
3.1 General Limits of Language to Logic Transformation

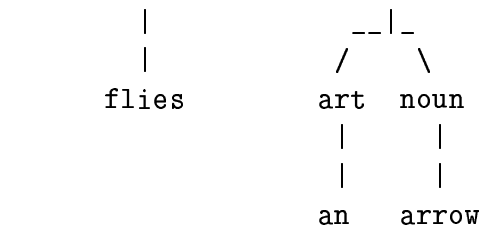
3.1.1 Ambiguity of the Part of Speech problem

Since English has an enormous amount of words that do not have an exact interpretation whether they are a noun, verb or anything else, we can get problems in parsing a sentence correct with our grammar. E.g.: "love" can be as well a noun as a verb. We will see now a few examples for ambiguous interpretation of a sentence. A very famous sentence to illustrate this problem is the following:

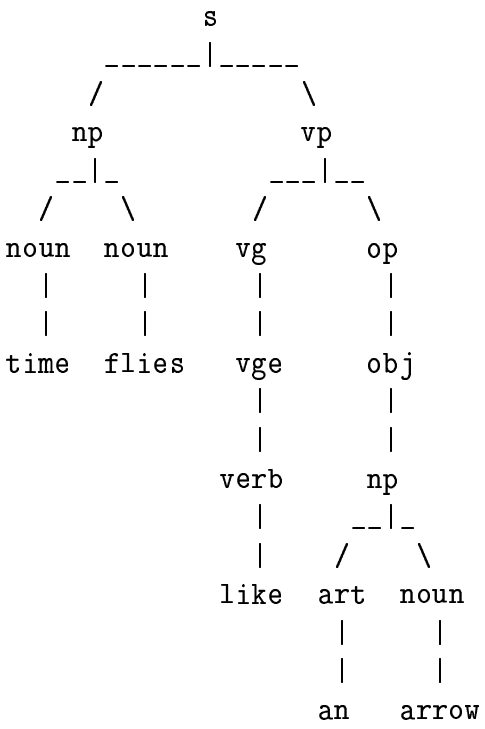
> Time flies like an arrow.

At first sight it seems quite clear that we have the following interpretation:





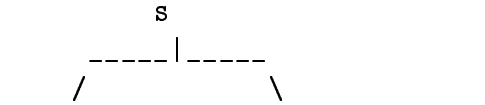
But if we look a bit closer, or try to parse it with a Parsing system we also get:

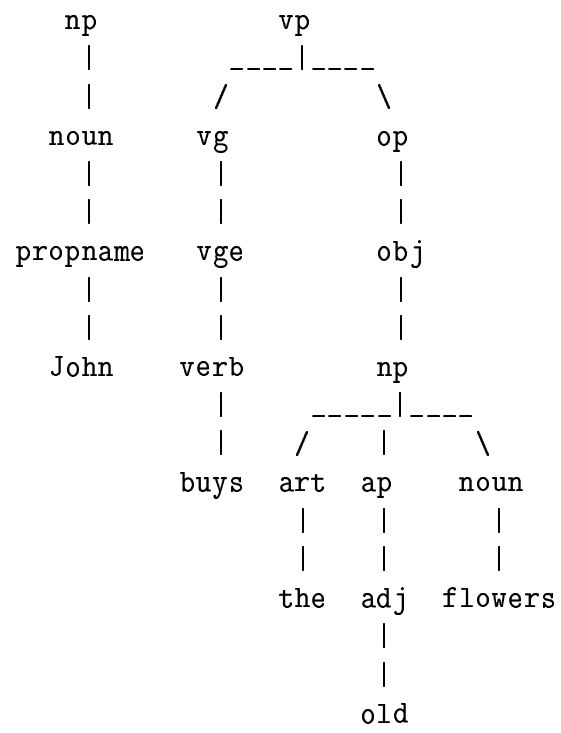


which seems a bit strange but has an semantic interpretation. We see that the ambiguity is caused by the possibility to interpret “flies” as plural of the noun “fly” or as a verbform of the verb “fly” and the uncertain role of “like” as verb or preposition.
The same effect can be seen if we parse:

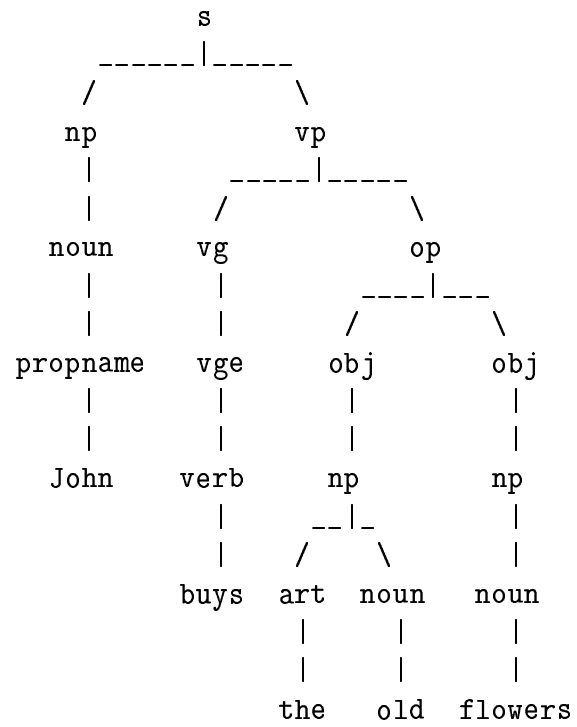
> John buys the old flowers.

Where we have the two interpretations:




np
vp
noun
vg
op
propname
vge
obj
John
verb
np
buys
art
ap
noun
the
adj
flowers
old

and


s
np
vp
noun
vg
op
propname
vge
obj
obj
John
verb
np
np
buys
art
noun
noun
the
old
flowers

which are caused by the possibility of "old" to be as well noun as adjective. There would be an enormous possibility for finding other examples. In general we can say that without knowing something about the context in which our sentence occurs we cannot say which parsing tree is the right one. And this leads us to different transformations into logic, which cannot be classified as wrong our true without knowing something else than our input.

3.1.2 New words are created every day

Besides the problem of detecting the correct part of speech of the words in our inputsentence, it may occur that we don't find this word in our dictionary. In the above algorithms we decided to define words as proper names, but of course this is to general and may cause problems. We could say that we would need a much bigger dictionary to have every existing word in it, but every spoken language is a living object an so is English too. E.g.: One might understand what is meant by christmasing, although it will not be found in any dictionary (at the moment when this text was written), but this can change very easily if the word becomes into fashion (something that happens every day)
So we can say that we will never have a 100% of concurrency between our dictionaries and the English used in reality.

3.1.3 Natural language is reflexive - First-order Logic is not

Even if we would solve the above problems and get every time a correct parsing tree, we would still have problems in transforming correct into First-order logic. E.g. if we want to transform the paradox sentence:

> This sentence is wrong.

we get

$$\exists x_1(this\_sentence(x_1) \wedge is\_wrong(x_1))$$

which of course loses the selfreference that we had in the input, since in First-order logic of predicates we cannot express such references . We would have to step to a logic of higher order to do this.
The same problem occurs if we want to transform sentences with direct or indirect speech. E.g.:

> Some say, that he is too old for this job.

Again we would need logic of higher order.
We also cannot transform inputsentences like questions and orders.

3.1.4 Ambiguity of Connectors and Quantifiers

We already have discussed this in 2.3.1 on page 12 and in 2.4.1 on page 23. It is obvious that the inexact use of connectors and quantifiers in natural language makes it impossible to translate them without losing some information or being inexact too.

3.2 Problems of the Algorithms presented above

Besides the general problems TPROP and TPRED do cause problems that might be solved by a better algorithm.

3.2.1 Treatment of Verbs

There are two main problems in TPRED when creating the predicates out of the verbs

- TPRED does not recognise the ending in "s" of verbs with third person subjects. This causes different predicates although they mean the same. E.g.: There can be predicates $love(x_i, x_j)$ and $loves(x_i, x_j)$

  An algorithm that solves this, would need a dictionary of verbforms for every verb. Then one of the first steps of this algorithm would be simply to change every third person verb by its indicative form.

- If we have different tenses we also get different predicates, when we would expect the same, since most verbs change if put in an other tense or even need an auxiliary verb. We can have: $will\_love(x_i, x_j)$, $loved(x_i, x_j)$, $has\_loved(x_i, x_j)$ and some other combinations when again we would only expect $love(x_i, x_j)$.

  To represent the different times logically correct we would have to extend our First-order Logic by a Temporal Logic, but if we want to avoid this, it is better to keep the time inside the predicate. This simple example shows us why:

  > If the traffic light is green, you can cross the street. The traffic light was green.

  Nobody given this information would cross the street, but if we eliminate the tense of our verbs we would get the logical implication that we can do this.

Therefore we put adverbs of time manner and place also into the predicate.

- If our verb is negated with "do not", "does not" or "did not". As the "not" is moved at the beginning of the sentence we get for example: $does\_love(x_i, x_j)$ instead of $love(x_i, x_j)$.

  For verbs in present tense this could be avoided easily by adding some grammar rules, but since we have the problem of the "s" still unsolved for lack of an adequate dictionary, we would still have a problem with that. We would lose the third person's "s" if we eliminate "does" and would not win anything. For the past tense we would have to change the negated verb into past tense which again is not possible without a dictionary of verbforms.

3.2.2 Pronoun Resolution

The greatest Problem TPRED is that it is not able to resolve pronouns (to find the nouns they refer to). E.g.:

Franz enjoys himself

If we transform this with TPRED we get:

$$enjoys(Franz, himself)$$

where we would expect:

$$enjoys(Franz, Franz)$$

or:

Vincent enters the restaurant and Jules watches him.

is transformed into

$$(\exists x_1(the\_restaurant(x_1) \wedge enters(Vincent, x_1)) \wedge (watches(Jules, him))$$

but we would expect

$$(\exists x_1(the\_restaurant(x_1) \wedge enters(Vincent, x_1)) \wedge (watches(Jules, Vincent))$$

And one final example:

Mark sells his house

becomes

$$\exists x_1(his\_house(x_1) \wedge sells(Mark, x_1))$$

Here we would expect maybe something like:

$$\exists x_1(house(x_1) \wedge owns(Mark, x_1) \wedge sells(Mark, x_1))$$

In [9] there can be found a intent to solve this by using Discourse Representation Theory (DRT), but since their algorithm bases on [8] which is designed for a small subset of English, it is yet to proof that it can be extended to LOGENG.

Chapter 4

An Implementation of TPROP and TPRED

Finally we will take a look at an implementation of the above described algorithms TPROP and TPRED called la2lo. Both algorithms are interpreted in one single program and performed one after the other using the same parsing data.

4.1 Description of la2lo

It is a PERL 5-program running under a LINUX or UNIX environment. The Perl part searches in the dictionaries, creates a Prolog-file containing the data found in the dictionaries and starts a PROLOG-interpreter. The files loaded by this interpreter were created under SWI-PROLOG 3.8.1 and do the parsing and the transformation.

4.1.1 Output format

Since la2lo only uses ASCII-text we have to express logical symbols with ASCII-signs. The following table shows their representation.

Symbol	ASCII-signs
$\neg$	`-`
$\wedge$	`&`
$\vee$	`|`
$\rightarrow$	`->`
$\leftarrow$	`<-`
$\exists$	`exists`
$\forall$	`all`

We will print expressions in First-order logic not only in the traditional line form but also in a tree representation, to make it easier to read. Section 4.2 will show a few examples of program output. It has the following format:

1. The input sentence
2. The version of this sentence used for parsing (may be slightly different
3. The results of the search in the dictionaries (The words of the sentence ordered alphabetically and followed by its part of speech, if it is ambiguous, we get more that one line with the same word) If the part of speech is connector we do not write it, since there is no Prolog-rule created. (This causes the words followed by nothing).
4. The parsing tree (If it exits, if not the program terminates.)
5. (optional) If `-a` option is set the question:

   Is the result of the parsing Ok for you (y/n)?

   If answered with "y" it leads to 6. If answered with "n", it is followed by the question:

   Do you want to continue the parsing (y/n)?

   Answer "n" terminates the program. Answer "y" continues the parsing and leads to point 4 of the output, if exists another parsing tree.
6. The result of the transformation into Propositional logic
7. The tree representation of the result of the transformation into First-order logic
8. The transformation into First-order logic
9. If `-a` option is set the question

   Do you want to continue the parsing (y/n)?

   with the same answering options as in 5. If the option is not set, the program continues as if it would have been answered with "y".

4.1.2 Description of the modules

We will see now a short description of the modules of la2lo.pl

la2lo.pl Perl-program, that is used to start the program, check the options (see below for details), read the input and start `search.pl`.

search.pl Perl-file, that splits the input into single sentences and searches for every sentence for its words in `moby.lex` and `link.lex` writes the results in `auto.pro` and according to the options set start the batchfile `mypl` or `myplfast`.

auto.pro Prolog-file created by `search.pl`, that includes additional rules for the grammar assigning the words occurring in the inputsentence to their part of speech and the starting rule for the Prolog-part.

mypl sh-batchfile, that starts a Prolog-interpreter with the following inputfiles `words.pro`, `tree.pro`, `conv2list.pro`, `conv2pred.pro` and `auto.pro` and starts this interpreter with the starting rule.

myplfast is the same as `mypl` but uses `words.pro` instead of `wordsfast.pro`

words.pro Prolog-file that includes the rules for the grammar used for the parsing and the dialog rules for the user-dialog. Starts `tree.pro` and `conv2list.pro`

wordsfast.pro Same as `words.pro` but with fewer grammar rules for faster parsing.

tree.pro Prolog-file slightly modified from [3] used for printing the parsing trees.

conv2list.pro Prolog-file includes the rules for TPROP, calls `conv2pred.pro`.

conv2pred.pro Prolog-file includes the rules for TPRED.

link.lex Dictionary used for la2lo. see description in section 2.1.1 on page 7

moby.lex Same as `link.lex`

Listings of most of the above files can be found in the appendix

4.1.3 How to start and install La2lo

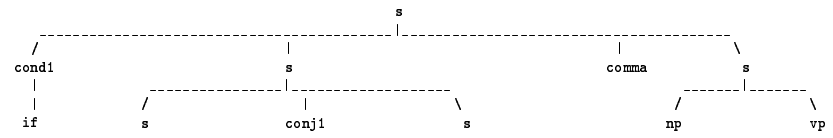
To install copy all the above files on a Unix-system containing a Perl-version 5.0 and a SWI-Prolog-version 3.8.1 or higher.
To start simply enter `latolo.pl [-hifa] [filename]`. Where `filename` is the name of the inputfile. If nothing specified `satz.txt` is taken.
The options have the following meaning:

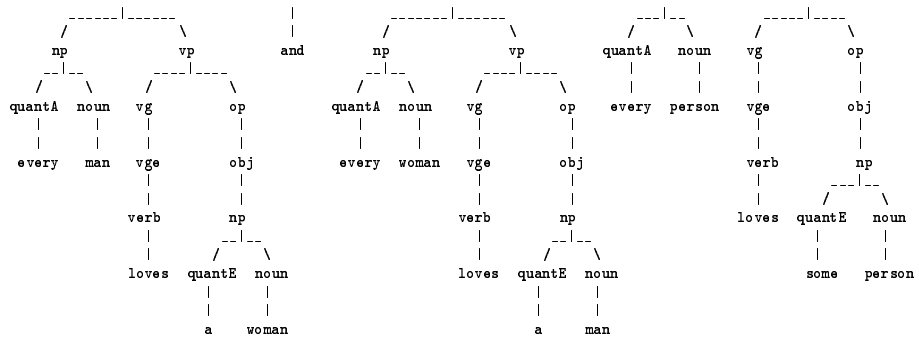
Option	Description
h	list a help text
i	interactive mode: The input sentence is read from STDIN The input of "exit" starts the parsing then.
f	fast parsing: The Program uses the simpler grammar in `words.pro`
a	user control of parsing: Since the parsing may be ambiguous the program asks the user after every parsing if the parsing tree is ok for him. And if so if he wants to continue the parsing after transforming this solution.

4.2 Examples

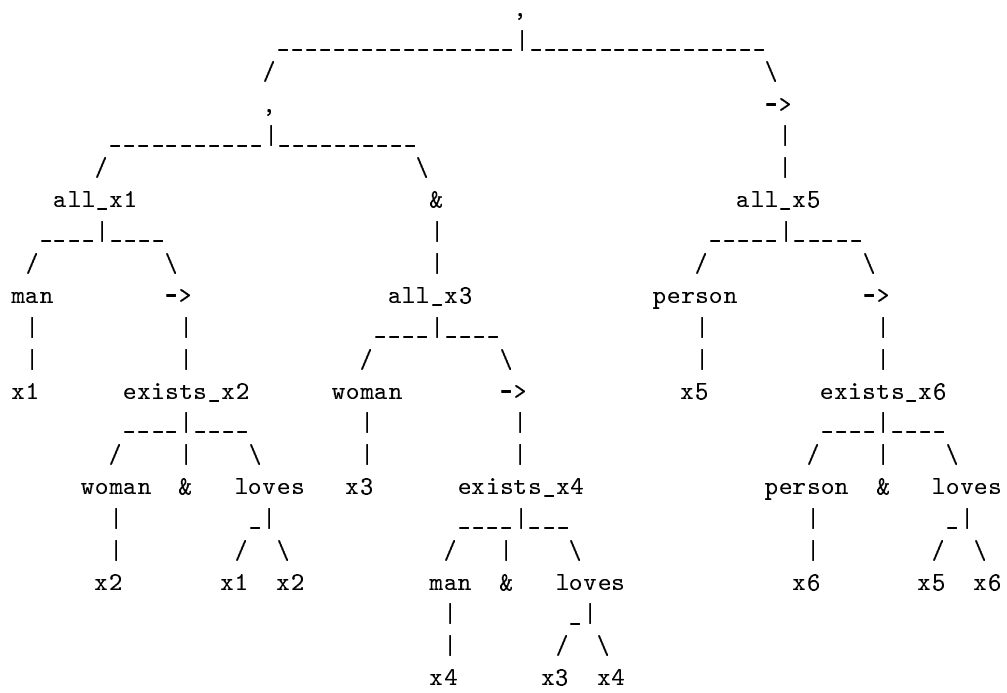
The following 2 examples have been created using only the option `-f` of latlo.

```
Fast Input:If every man loves a woman and every woman loves a
man, everybody loves somebody.

sentence:If every man loves a woman and every woman loves a man
'(,)'  every person loves some person
'(,)'
and
if
loves verb
man
man adj
man noun
man verb
person noun
woman noun
woman verb
```






```
(((every man loves a woman) & (every woman loves a man))
-> (every person loves some person))
```

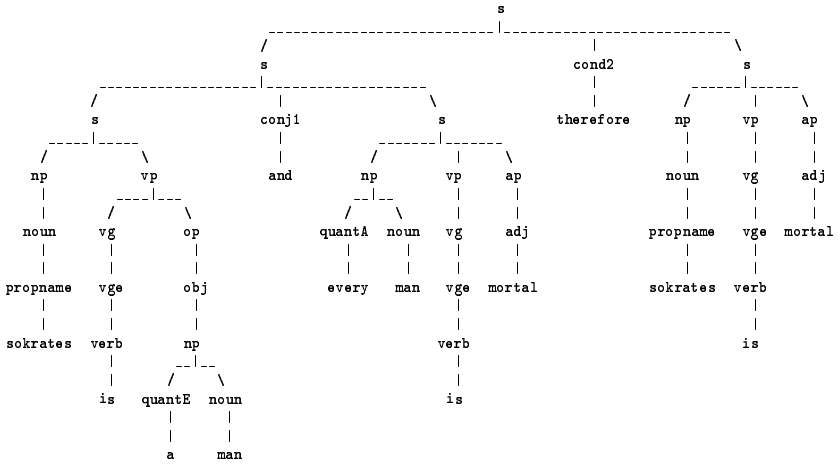



```
((all x1 (man(x1) -> (exists x2 (woman(x2) & loves(x1,x2)))))
& (all x3 (woman(x3) -> (exists x4 (man(x4) & loves(x3,x4))))))
-> (all x5 (person(x5) -> (exists x6 (person(x6)
& loves(x5,x6)))))
```

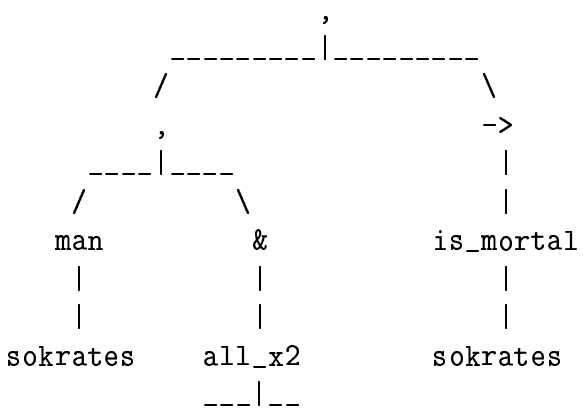
The next example is the classic logical sentence.

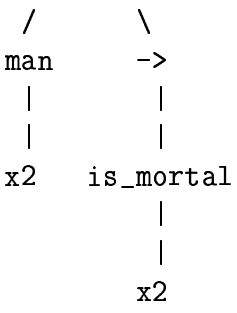
```
Input:Sokrates is a man and every man is mortal
therefore sokrates is mortal.
```

```
sentence:Sokrates is a man and every man is mortal
therefore sokrates is mortal
and
is verb
man
man adj
man noun
man verb
mortal adj
mortal noun
sokrates propname
therefore
```




```
(((sokrates is a man) & (every man is mortal))
-> (sokrates is mortal))
```

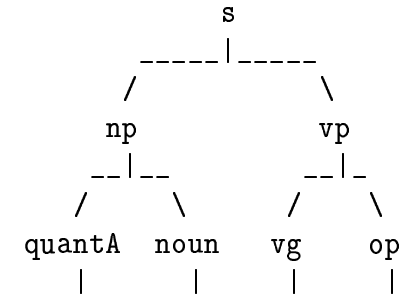


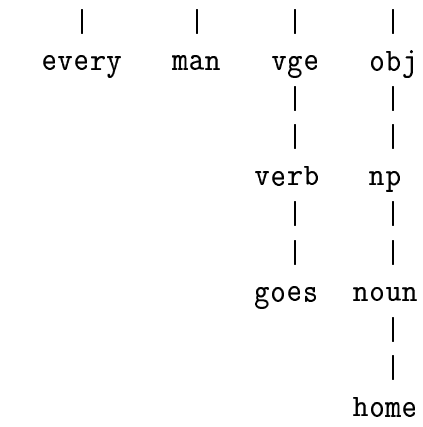



```
((man(sokrates)) & (all x2 (man(x2) -> (is_mortal(x2)))))
-> (is_mortal(sokrates)
```

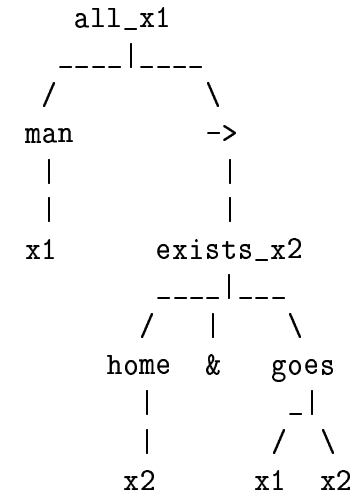
The next example demonstrates la2lo started with the options `-fia`.

```
Fast Ask Userinput La2Lo Language to logic

  Input your sentences (and close with exit)
 :Every man goes home.
exit
Input:Every man goes home.

sentence:Every man goes home
goes noun
goes verb
home adj
home noun
home verb
home adv
man
man adj
man noun
man verb
```

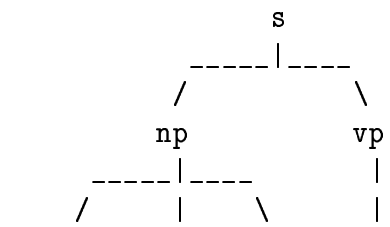


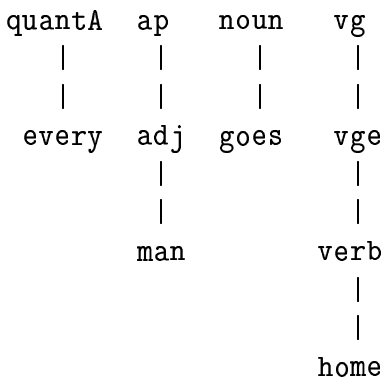



```
Is the result of the parsing Ok for you (y/n)?y

(every man goes home)
```

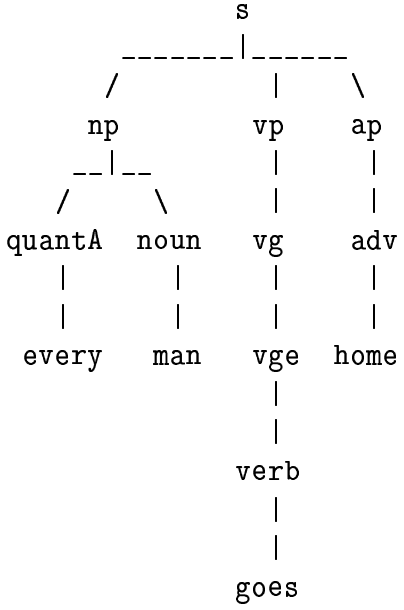



```
all x1 (man(x1) -> (exists x2 (home(x2) & goes(x1,x2))))

Do you want to continue the parsing (y/n)?y
```

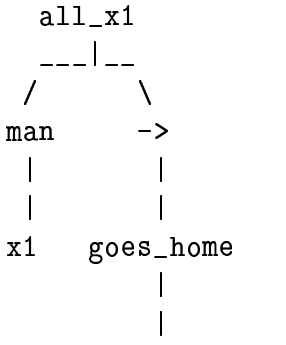




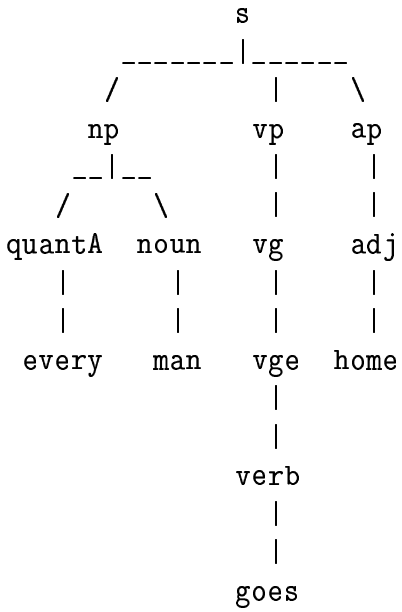

```
Is the result of the parsing Ok for you (y/n)?n
```




```
(every man goes home)
```



```
         x1

all x1 (man(x1) -> (goes_home(x1)))

Do you want to continue the parsing (y/n)?y
```

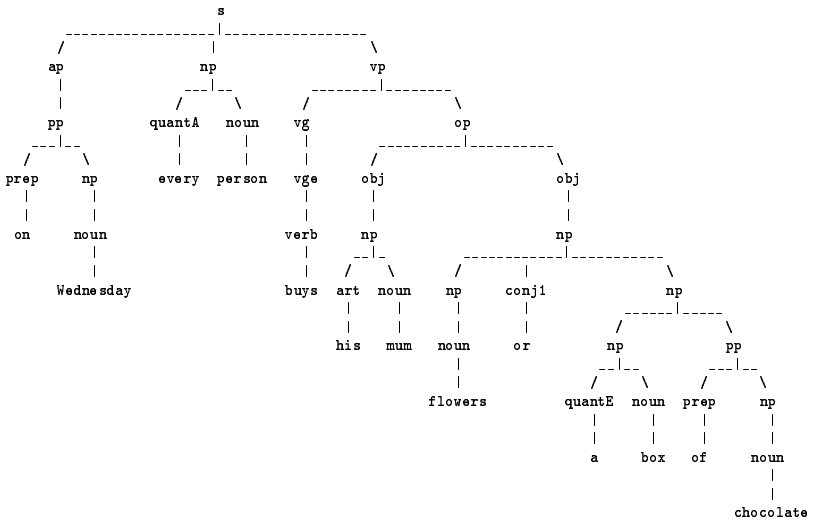



```
Is the result of the parsing Ok for you (y/n)?n
```

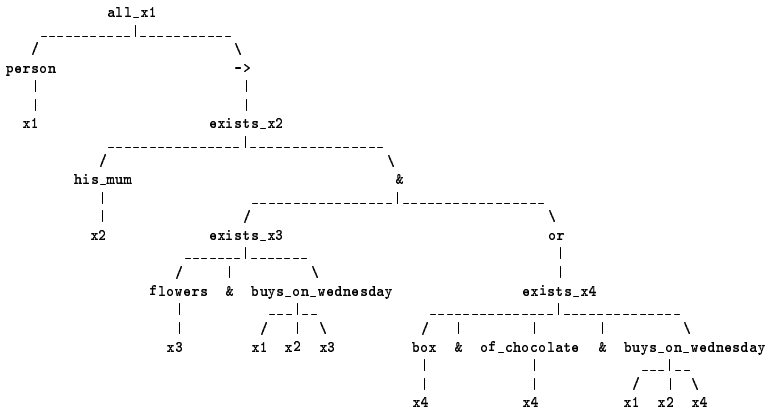
In the last example we can see, the problem of ambiguous parsing:

```
Input:On Wednesday everyone buys his mum flowers
or a box of chocolate.

sentence:On Wednesday every person buys his mum flowers
or a box of chocolate
Wednesday noun
box noun
box verb
buys verb
chocolate noun
flowers noun
flowers verb
his art
his pron
```

```
mum adj
mum noun
mum verb
of prep
on adj
on noun
on prep
on adv
or
person noun
```

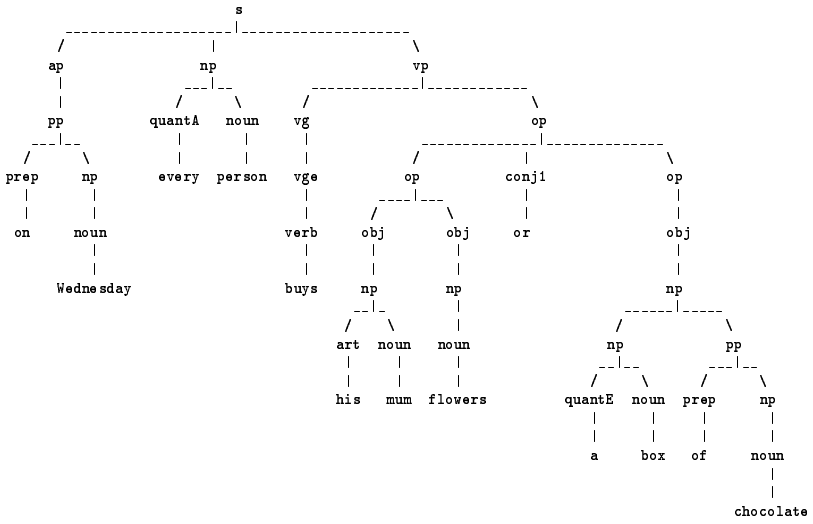



```
((on Wednesday every person buys his mum flowers) |
(on Wednesday every person buys his mum a box of chocolate))
```

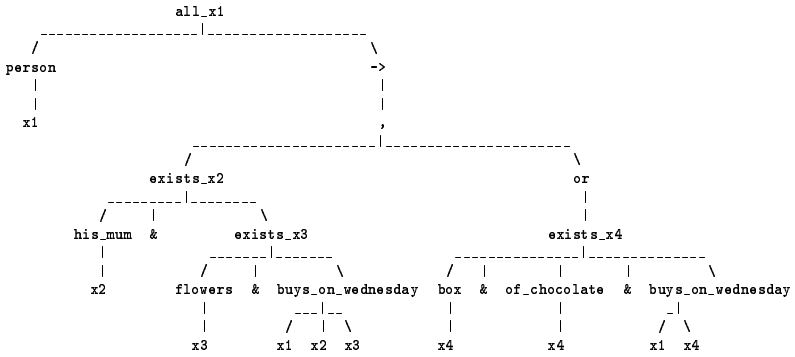



```
all x1 (person(x1) -> (exists x2 (his_mum(x2) &
```

```
((exists x3 (flowers(x3) & buys_on_Wednesday(x1,x2,x3))) |
(exists x4 (box(x4) & of_chocolate(x4) &
buys_on_Wednesday(x1,x2,x4)))))))
```




```
((on Wednesday every person buys his mum flowers) |
(on Wednesday every person buys a box of chocolate))
```




```
all x1 (person(x1) -> (((exists x2 (his_mum(x2) &
exists x3 (flowers(x3) & buys_on_Wednesday(x1,x2,x3))))
| (exists x4 (box(x4) & of_chocolate(x4) &
buys_on_Wednesday(x1,x4))))))
```

Appendix A

Grammar

The following grammar in BNF was used to do the parsing. (For better reading we use `A --> B` and `A --> C` instead of `A --> B|C`). Terminating symbols are surrounded by `[ ]`.

```
comma --> [,]
then  --> [then]
else  --> [else]
or    --> [or]
and   --> [and]
no    --> [no]
to    --> [to]
either --> [either]
neg   --> [not]

%auxiliary verbs

be   --> [be]
been --> [been]

have --> [have]
has  --> [has]
had  --> [had]
do   --> [do]
does --> [does]
did  --> [did]

vb --> [am]
vb --> [is]
```

```
vb --> [are]
vb --> [was]
vb --> [were]

conj1 --> comma
conj1 --> [and]
conj1 --> [or]

conj2 --> [although]

conj3 --> [but]
conj3 --> [however]

cond1 --> [if]
cond1 --> [as]
cond1 --> [because]
cond1 --> [since]

cond2 --> [therefore]
cond2 --> [thus]
cond2 --> [so]

%relative pronouns

rpn1 --> [who]
rpn1 --> [whom]
rpn1 --> [which]
rpn1 --> [that]
rpn2 --> [whose]
rpn2 --> [what]

%quantifiers
quantA --> [every]
quantA --> [all]
quantA --> [each]

quantE --> [some]
quantE --> [one]
quantE --> [a]
quantE --> [an]
```

```
%verbs
verbgroupend --> verbgroupend,conj1,verbgroupend
verbgroupend --> verbgroupend,conj3,verbgroupend
verbgroupend --> verbgroupend,to,verb
verbgroupend --> verbgroupend,gerund
verbgroupend --> verb
verbgroupend --> adv,verb

%passive
verbgroupend --> been,verb
verbgroupend --> adv,been,verb
verbgroupend --> be,verb
verbgroupend --> adv,be,verb

verbgroup --> verbgroup,conj1,verbgroup
verbgroup --> verbgroup,conj3,verbgroup
verbgroup --> verbgroup,prep
verbgroup --> verbgroup,to

verbgroup --> verbgroupend
verbgroup --> have,verbgroupend
verbgroup --> have,neg,verbgroupend
verbgroup --> has,verbgroupend
verbgroup --> has,neg,verbgroupend
verbgroup --> had,verbgroupend
verbgroup --> had,negverbgroupend

verbgroup --> do,verbgroupend
verbgroup --> do,neg,verbgroupend
verbgroup --> does,verbgroupend
verbgroup --> does,neg,verbgroupend
verbgroup --> did,verbgroupend
verbgroup --> did,neg,verbgroupend

verbgroup --> mv,verbgroupend
verbgroup --> mv,neg,verbgroupend
verbgroup --> mv,have,verbgroupend
verbgroup --> mv,neg,have,verbgroupend

%is,are,was,were,
verbgroup --> vb,gerund
```

```
verbgroup --> vb,neg
verbgroup --> vb,neg,gerund
verbgroup --> vb,neg,verb
verbgroup --> vb,adv
verbgroup --> vb,adv,gerund
verbgroup --> vb,adv,verb
verbgroup --> vb,neg,adv
verbgroup --> vb,neg,adv,gerund
verbgroup --> vb,neg,adv,verb

adjectivephrase --> adjectivephrase,conj1,adjectivephrase
adjectivephrase --> adjectivephrase,conj3,adjectivephrase
adjectivephrase --> adj

%objects, subjects, relative clauses

nounphrase --> nounphrase relativeclause

%negation
nounphrase --> no,nounphrase

nounphrase --> nounphrase,nounphrase
nounphrase --> nounphrase,prepphrase

nounphrase --> noun
nounphrase --> quantA,noun
nounphrase --> quantE,noun
nounphrase --> pron
nounphrase --> art,noun
nounphrase --> adjectivephrase,noun
nounphrase --> quantA,adjectivephrase,noun
nounphrase --> quantE,adjectivephrase,noun
nounphrase --> art,adjectivephrase,noun

prepphrase --> prep,nounphrase
prepphrase --> to,nounphrase

adverbphrase --> adverbphrase,adverbphrase
adverbphrase --> prepphrase
adverbphrase --> adv
```

```
%objects
objectphrase --> object
objectphrase --> object,object
objectphrase --> objectphrase,conj1,objectphrase

object--> nounphrase

%verbphrase
verbphrase --> verbgroup,relativeclause
verbphrase --> verbphrase,conj1,verbphrase
verbphrase --> verbphrase,conj3,verbphrase
verbphrase --> verbgroup
verbphrase --> verbgroup, objectphrase


% relative clauses
relativeclause --> relativeclause,conj1,relativeclause
relativeclause --> rpn1, verbphrase
relativeclause --> rpn1, sentence
relativeclause --> rpn2, sentence

% what to do.
relativeclause --> rpn2, to, verb

relativeclause --> comma, sentence
relativeclause --> comma, sentence,comma

%sentences

% ... and therefore ...
sentence --> sentence,and,cond2,sentence

% and, or, ,
sentence --> sentence,conj1,sentence
% although
sentence --> sentence,conj2,sentence

% but, however
sentence --> sentence,conj3,sentence
```

```
% therefore
sentence --> sentence,cond2,sentence

% if, as,..
sentence --> sentence,cond1,sentence
sentence --> cond1,sentence,then,sentence
sentence --> cond1,sentence,comma,sentence
sentence --> cond1,sentence,then,sentence,else,sentence
sentence --> cond1,sentence,then,sentence,or,else,sentence

%either .. or
sentence -->  either,sentence,or, sentence

%standard sentence
sentence -->  nounphrase,verbphrase
sentence -->  adverbphrase,nounphrase,verbphrase
sentence -->  nounphrase,verbphrase,adverbphrase
sentence -->  nounphrase,verbphrase,adjectivephrase
sentence -->  nounphrase,verbphrase,adjectivephrase,adverbphrase
```

Appendix B

Listing of the programs

B.1 la2lo.pl

```
#!/usr/bin/perl

# MAIN PROGRAM  OF LATOLO
#
#       EXECUTABLE VERSION
#
# V 1.0
#
# main program see search.pl
#
#

sub readin
{# read in the file of natural language sentences
$inputs="";
open(textdatei,"<".$inputfile) ||
die "Cannot open the input file ",$inputfile, " .\n";

while (<textdatei>) { # for each line in textdatei
  s/\#.+//g;          # Ignore comments after an #
  s/\n/ /gi;          # delete the returns
  $inputs=$inputs.$_; # concat to one string
}
close(textdatei);
print "Input:",$inputs,"\n";


}

sub readstdin {
       print " La2Lo Language to logic \n\n ";
```

```
       print " Input your sentences (and close with exit) \n :";
       $inputs="";
       while (( defined( $_ =  <STDIN>))  && not (/exit/)) {
              s/\#.+//g;          # Ignore comments after an #
              s/\n/ /gi;          # delete the returns
              $inputs = $inputs.$_;
         }

}

sub parseargs
{
$speed = 0;
$ask   = 0;
if ( $#ARGV == -1)
{
       $inputfile = "satz.txt";
       print "no argument, taking '", $inputfile,"' \n";
       readin;
}
else
{
   if ($ARGV[0]=~ /\-[hcghcbb]/)
   {
       print " La2Lo Language to logic \n\n ";
       print " Usage: la2lo.pl [-hifa] [filename]\n\n" ;
       print " -i interactive mode\n";
       print " -a ask if parsing ok\n";
       print " -f fast parsing (for simple senteces)\n";
       print " filename input text to convert to logic\n";
   }
   else
   {
       if ($ARGV[0]=~ m/^-/)
       {
       if (($ARGV[0]=~ m/[0123456789bcdegjklmnopqrstuvxyz]/) ||
              ($ARGV[0]=~ m/[ABCDEFGHIJKLMNOPQRTSTUVWXYZ]/))
          {
           print "ilegal options\n";
            print " Usage: la2lo.pl [-h?ifa] [filename]\n\n" ;
          }
          else
          {
           if ($ARGV[0]=~ m/f/)
          {
            print "Fast ";
               $speed = 1;
         }
```

```
          if ($ARGV[0]=~ m/a/)
          {
            print "Ask ";
               $ask = 1;
           }
           if ($ARGV[0]=~ m/i/)
           {
               print "Userinput";
               readstdin;
           }
           else
           {
                 if ( $#ARGV == 0)
                 {
                    $inputfile = "satz.txt";
                    print "no argument, taking '", $inputfile,"' \n";
                    readin;
                 }
                 else
                 {
                 $inputfile = $ARGV[1];
                 readin;
                 }
           }
          }
       }
       else
       {
$inputfile = $ARGV[0];
            readin;
       }

   }

}

}

parseargs;
print "Input:",$inputs,"\n";

$defoutput = STDOUT;
#import search.pl where the main work is done.
do "search.pl" ;#|| die "search.pl not found";
```

endverbatim

B.2 search.pl

```
# MAIN SUBS  OF LATOLO
# V 1.0
#
# The parts of the program
#
# o Translation Words to Prolog
# o Preprocessing of the input words
# o Seek in the wordlist
# o main program
# o references

#-----------------------------------------------------------------
#----------------------Translation Words to Prolog---------------
#-----------------------------------------------------------------

sub pred{#($_:wordart):converted wordart;
  s/!//; # to avoid problems with wor(d)
  s/d/D/g; # to avoid problems with wor(d)
  s/a/A/g; # to avoid problems with (a)rt
  s/v/Y/g; # to avoid problems with (v)erb
  s/r/R/g; # to avoid problems with ve(r)b

  s/[Nhmp]/noun/;
  s/[Vtliq]/verb/;
  s/[ID]/art/;
  s/C//g;
  s/x/propname/;
  s/Y/adv/;
  s/P/prep/;
  s/A/adj/;
  s/R/pron/;
  s/M/mv/;
  s/G/gerund/;
  return $_
}
sub convertwordart #($_[0] : wordart):converted wordart;
                   # Returns the type of the word (wordart)
{
  $_=$_[0];
  s/\n//g;     #Delete returns
  s/.*g.*/G/;  #if it is a Gerund ...
  s/p/N/;
  s/e/v/;
  s/c/C/;
  s/a/A/;
  s/[^D]*r[^D]*/r/;
  #if it is a pronoun and not also an article,
```

```
  #it is only a pronoun
  tr/[Vitlq]/V/s;
  tr/[Nhmp]/N/s;
  tr/[IDd]/D/s;
  my @list;
  my $help;
  @list = split(//,$_);
  @list=delsame(sort(@list));

  for (@list)
  {
     $help=$help.$_;
  }
  $_=$help;

  s/[^VP]*C[^VP]*/C/;
  #if conjunction and not also a verb, only a conjunction

  if ($_ eq "") {$_="x";}
  if (/M/)
  {
   s/[Vitlq]//;#
   s/[Vitlq]//;#
  }
  return $_;
}

#For all words prints the prolog format via out()
sub prologout#();
{
my $i; #REF 2
 $verbsP = "";
 $nounsP = "";
 $adjP = "";
 $advP  = "";  #adverb
 $prepP  = ""; #preposition
 $artP = "";
 $pronP = ""; #pronoun
 $propP = "";
 $mvP   = ""; #modalverbs
 $gerundP = "";
 for($i=0;$i <= $#words ;$i++) {
       $conv = &convertwordart($wordart[$i]);
      @wart=split(//,$conv);
      # split the grammatical type(s) (Wortart)
      for (@wart) {
         pred($_);
         print $words[$i]," ",$_,"\n";
```

```
        out($words[$i],$_);
       }
  }
}


sub  out #($_0: wort; $_1 :wortart);
      # output  of a single word in prolog format
{

   if ($_[1] ne "") {   #if not empty

     # output format :
     #  E.g: verb(verb(eat)) --> [eat].
     #

      if ($_[0]=~/\A[ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789]/)
     # quote uppercase words to pervent Prolog
     # from treating them as Variables
     # \A is just like ^
          {  $_[0]="'".$_[0]."'"; }

if ($_[1] eq "noun")
{$nounsP= $nounsP. $_[1]. '(' .$_[1].
          '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ;}
else {if ($_[1] eq "verb")
{$verbsP= $verbsP. $_[1]. '(' .$_[1].
          '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ;}
else {if ($_[1] eq "adj")
{$adjP= $adjP. $_[1]. '(' .$_[1].
         '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "art")
{$artP= $artP. $_[1]. '(' .$_[1].
         '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "adv")
{$advP= $advP. $_[1]. '(' .$_[1].
         '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "prep")
{$prepP= $prepP. $_[1]. '(' .$_[1].
          '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "pron")
{$pronP= $pronP. $_[1]. '(' .$_[1].
          '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "mv")
{$mvP= $mvP. $_[1]. '(' .$_[1].
         '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
else {if ($_[1] eq "gerund")
{$gerundP= $gerundP. $_[1]. '(' .$_[1].
         '('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
```

```
else {if ($_[0] ne "")
{$propP= $propP.
 'propname(propname('.$_[0].')) --> ['. $_[0]."]. %\t"."\n" ; }
                                                              }
                                                          }
                                                      }
                                                  }
                                              }
                                          }
                                      }
                                  }
                              }
                          }
     }

}

#-----------------------------------------------------------------
#--------------Preprocessing of the input words------------------
#-----------------------------------------------------------------

sub delsame #(@_:list):list;
#deletes entries that appear more than once,
#presumes a sorted list
{
 my $wg;     # the last element
 my @warray;

 @warray=@_;#The first and only parameter of our function

 for(@warray)
 {
  if (($wg eq $_) || ($_ eq "every") || ($_ eq "any")  ||
                     ($_ eq "some")  || ($_ eq "a")    ||
                     ($_ eq "all")   || ($_ eq "each") ||
                     ($_ eq "one")   || ($_ eq "an"))
  {$_="zzzzzz";} #Set to, hopefully, the last element.
  else
  {$wg=$_;}       #last = current
 }

 @warray = sort(@warray);
 # Now the zz...'s that we want to delete are at the end
 do { $wg=pop(@warray)
 }while ($wg eq "zzzzzz");  # Delete them one by one
 push(@warray,$wg);
 # Add the last element we deleted inadvertedly
 return @warray; # return the value

}
```

```
sub replacent#($_:word):converted word;   Replace n't;

{
s/i\'m/i am/gi;
s/you\'re/you are/gi;
s/he\'s/he is/gi;
s/she\'s/she is/gi;
s/they\'re/they are/gi;
s/we\'re/we are/gi;

s/isn\'t/is not/g;
s/aren\'t/are not/g;
s/were\'nt/were not/g;
s/ain\'t/is not/g;

s/don\'t/do not/g;
s/doesn\'t/does not/g;
s/didn\'t/did not/g;

s/i\'ve/i have/gi;
s/we\'ve/we have/gi;
s/you\'ve/you have/gi;
s/they\'ve/they have/gi;


s/haven\'t/have not/g;
s/hasn\'t/has not/g;
s/hadn\'t/had not/g;


s/i\'ll/i will/gi;
s/you\'ll/you will/gi;
s/he\'ll/he will/gi;
s/she\'ll/she will/gi;
s/it\'ll/it will/gi;
s/we\'ll/we will/gi;
s/they\'ll/they will/gi;

s/i\'ld/i would/gi;
s/you\'ld/you would/gi;
s/he\'ld/he would/gi;
s/she\'ld/she would/gi;
s/it\'ld/it would/gi;
s/we\'ld/we would/gi;
s/they\'ld/they would/gi;


s/won\'t/will not/g;
```

```
s/wouldn\'t/would not/g;

s/mustn\'t/must not/g;

s/cannot/can not/g;
s/can\'t/can not/g;
s/couldn\'t/could not/g;

s/shan\'t/shall not/g;
s/shouldn\'t/should not/g;

s/mightn\'t/might not/g;

s/neither/no/g;
s/nor/and no/g;

#for first order logic

s/nobody/not somebody/g;
s/nothing/not something/g;

s/everybody/every person/g;
s/everyone/every person/g;
s/everything/every thing/g;
s/anybody/every person/g;
s/anyone/every person/g;
s/anything/every thing/g;

s/someone/some person/g;
s/somebody/some person/g;
s/something/some thing/g;

s/any/every/g;

s/Nobody/not somebody/g;
s/Nothing/not something/g;

s/Everybody/every person/g;
s/Everyone/every person/g;
s/Everything/every thing/g;
s/Anybody/every person/g;
s/Anyone/every person/g;
s/Anything/every thing/g;

s/Someone/some person/g;
s/Somebody/some person/g;
s/Something/some thing/g;

s/Any/every/g;
```

```
return $_;
}
#-------------------------------------------------------------
#--------------- Seek in the  lexikon (wordlist)--------------
#-------------------------------------------------------------

# sequentially seeks in the wordlist
sub seq  #();
{
my $seektwice=0;
my $lexe; # entry of the lexikon; 0:word 1:wordart
my $i=0;

 print defoutput "Seek in the wordlist\n";
 while (<lexikon>)
 { # for each entry of the wordlist
   # split the entry into a word and a wordart part
   @lexe = split(/$delim/, $_);


      if (($i <= $#words ) && ($words[$i] le $lexe[0]))
         # if there is any word left and
         # if  the lexikon entry is bigger

      {
         if (($words[$i] eq $lexe[0]))
         {# If the entry matches our current word
                  $wordart[$i] = $lexe[1].$wordart[$i];
         }
         else
         {  if ($words[$i] ne "'(,)'")
            {# (,) may also be added to the wordlist
                 #not found, currently do nothing
            }
            else {$wordart[$i]="C"}
         }
         $wordart[$i] =~s/\n//g;
         #Seek each word twice to get all possible types.
         if ($seektwice == 1)
                {$i++;
                 $seektwice=0 #;
                }
         else
                { $seektwice = 1;
                }

       }
```

```
 }

}
#-------------------------------------------------------------
sub findlex{  #(); Seeks via seq in both of our wordlists
my $i;
my $empty;
    for($i=0;$i <= $#words; $i++) {
    $wordart[$i] = $empty;}

   open(lexikon,"<moby.lex") ||
       die "wordlist moby.lex not found";
    seq;
   close(lexikon);
   open(lexikon,"<link.lex") ||
       die "wordlist link.lex not found";
    seq;
   close(lexikon);
   }
sub joinprolog#(@_:list):string;
{
 my @inpu;
 @inpu=@_;
 for (@inpu)  {
    if (/\A[ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789]/)
    # quote uppercase words to pervent
    # Prolog from treating them as Variables
        {  $_="'".$_."'"; }

   }
 return join(',',@inpu);

}
#-------------------------------------------------------------
#-----------------------  main program -----------------------
#-------------------------------------------------------------

{
my @saetze;
my $prologinput;

#delimiter ASCII 215 of our format of the wordlist
$delim=chr(215);


@saetze = split(/\./,$inputs);
#get the sentences, they are
#seperated by "." (saetze = sentences)
```

```
for (@saetze) {#for each sentence
  s/[\,\;]/ \'\(\,\)\' /gi;# Convert , to (,)
  s/^ //;    # Delete spaces at the beginning.
  s/  / /gi; # Two spaces to one one.
  $_=replacent($_); # "Don't" to "do not",...
  print $defoutput "\n"."sentence:", $_,"\n";
  $help = $_;
  # get the words, they are seperated by space
  @words = split(/ /,$_);
   # Note: we already converted the comma


  if ($#words != -1) { #if the sentence is not empty
    if ($words[0] ne "I") {$words[0] =lc($words[0]);}
    # Convert first word   of the sentence into lowercase.
    # Note there are 2 possibilities: I may be a Propname
    # or it may only be uppercased because it
    # stands on the beginning of the sentence

   $depth = int (($#words+1) / 2);

   if ($depth < 3) {$depth = 3;}
   else {if ($depth > 5) {$depth = 5;}}

   if ($ask == 0)
   {
   $prologinput =
   "main :- multitranslate([".joinprolog(@words)."],".$depth.").";
   }
   else
   {
   $prologinput =
   "main :- asktranslate2([".joinprolog(@words)."],".$depth.")."
   }
   @words = sort(@words);
   # deletes entries that appear more than once,
   # presumes a sorted list
   @words = delsame(@words);

   findlex;  # For each $words[$i] get the  $wordart[$i];

   open(outfile,">auto.pro");
    prologout;     # output of the words in prolog format
    #the folowing ifs avoid empty predicates in prolog
    if ($verbsP eq "")
     {$verbsP=  "verb(_,_,_) :- fail. %\t"."\n" ; }
    if ($gerundP eq "")
{$gerundP=  "gerund(_,_,_) :- fail. %\t"."\n" ; }
    if ($mvP eq "")
```

```
{$mvP=  "mv(_,_,_) :- fail. %\t"."\n" ; }
    if ($adjP eq "")
{$adjP=  "adj(_,_,_) :- fail. %\t"."\n" ; }
    if ($advP eq "")
{$advP=  "adv(_,_,_) :- fail. %\t"."\n" ; }
    if ($prepP eq "")
{$prepP=  "prep(_,_,_) :- fail. %\t"."\n" ; }
    if ($pronP eq "")
{$pronP=  "pron(_,_,_) :- fail. %\t"."\n" ; }
    if ($artP eq "")
       {$artP=  "art(_,_,_) :- fail. %\t"."\n" ; }
    if ($propP eq "")
{$propP=  "propname(_,_,_) :- fail.","\n"; }
    print outfile '%------------------------','\n";
    print outfile $prologinput, "\n";
    print outfile "\n";
    print outfile '%------------------------','\n";
    print outfile $verbsP;
    print outfile $mvP;
    print outfile $gerundP;
    print outfile "\n";
    print outfile $adjP;
    print outfile "\n";
    print outfile $advP;
    print outfile "\n";
    print outfile $prepP;
    print outfile "\n";
    print outfile $pronP;
    print outfile "\n";
    print outfile $artP;
    print outfile "\n";
    print outfile $nounsP;
    print outfile 'noun(noun(N)) --> propname(N).','\n";
    print outfile "\n";
    print outfile $propP;
    close(outfile);
     if ( $speed == 1)
     {
     system("myplfast");
     }
     else
     {
     system("mypl");
     }

     system("l2n.out");
 }
 }
```

```
}

return 1;
```

endverbatim

B.3 auto.pro

```
%-------------------------
main :- multitranslate([sokrates,is,a,man,and,every,
man,is,mortal,therefore,sokrates,is,mortal],5).
%-------------------------
verb(verb(is)) --> [is].
verb(verb(man)) --> [man].
mv(_,_,_) :- fail.
gerund(_,_,_) :- fail.

adj(adj(man)) --> [man].
adj(adj(mortal)) --> [mortal].

adv(_,_,_) :- fail.

prep(_,_,_) :- fail.

pron(_,_,_) :- fail.

art(_,_,_) :- fail.

noun(noun(man)) --> [man].
noun(noun(mortal)) --> [mortal].
noun(noun(N)) --> propname(N).

propname(propname(sokrates)) --> [sokrates].
```

B.4 mypl and myplfast

B.4.1 mypl

```
#!/bin/sh
pl -g main. -o l2n.out -c words.pro tree.pro auto.pro
conv2list.pro conv2pred.pro 2> /dev/null
```

B.4.2 myplfast

```
#!/bin/sh
pl -g main. -o l2n.out -c wordsfast.pro tree.pro auto.pro
conv2list.pro conv2pred.pro 2> /dev/null
```

B.5 words.pro and wordsfast.pro

The rules not used in `wordsfast.pro` are followed by `% only word.pro`

```
comma --> [’(,)’].
then  --> [then].
else  --> [else].
or    --> [or].
and   --> [and].
no --> [no].
to --> [to].
either --> [either].
neg(neg(not)) --> [not].

%auxilary verbs

be --> [be].
been --> [been].

have --> [have].
has --> [has].
had --> [had].
do --> [do].
does --> [does].
did --> [did].

vb(vb(am)) --> [am].
vb(vb(is)) --> [is].
vb(vb(are)) --> [are].
vb(vb(was)) --> [was].
vb(vb(were)) --> [were].

conj1(conj1(’(,)’)) --> comma.
conj1(conj1(and)) --> [and].
conj1(conj1(or)) --> [or].

conj2(conj2(although)) --> [although].

conj3(conj3(but)) --> [but].
conj3(conj3(however)) --> [however].

cond1(cond1(if)) --> [if].
cond1(cond1(as)) --> [as].
cond1(cond1(because)) --> [because].
cond1(cond1(since)) --> [since].

cond2(cond2(therefore)) --> [therefore].
cond2(cond2(thus)) --> [thus].
cond2(cond2(so)) --> [so].
```

```
%relative pronouns
rpn1(rpn1(who))   --> [who].
rpn1(rpn1(whom))  --> [whom].
rpn1(rpn1(which)) --> [which].
rpn1(rpn1(that))  --> [that].
rpn2(rpn2(whose)) --> [whose].
rpn2(rpn2(what))  --> [what].

%quantifiers
quantA(quantA(every)) --> [every].
quantA(quantA(all)) --> [all].
quantA(quantA(each)) --> [each].

quantE(quantE(some)) --> [some].
quantE(quantE(one)) --> [one].
quantE(quantE(a)) --> [a].
quantE(quantE(an)) --> [an].

%verbs
verbgroupend(vge(VG1,C,VG2),L)  -->{Y is L+1, Y =< 3 },
    verbgroupend(VG1,Y),conj1(C),verbgroupend(VG2,Y).
verbgroupend(vge(VG1,C,VG2),L)  -->{Y is L+1, Y =< 3 },
    verbgroupend(VG1,Y),conj3(C),verbgroupend(VG2,Y).
verbgroupend(vge(VG,to,V),L)   -->{Y is L+1, Y =< 3 },
    verbgroupend(VG,Y),to,verb(V).
verbgroupend(vge(VG,G),L)   -->{Y is L+1, Y =< 3 },
    verbgroupend(VG,Y),gerund(G).
verbgroupend(vge(V),_)          --> verb(V).
verbgroupend(vge(ADV,V),_)      --> adv(ADV),verb(V).

%passive
verbgroupend(vge(been,V),_)     --> been,verb(V).% only word.pro
verbgroupend(vge(been,ADV,V),_) --> adv(ADV),been,verb(V).% only word.pro
verbgroupend(vge(be,V),_)       --> be,verb(V).% only word.pro
verbgroupend(vge(be,ADV,V),_)  --> adv(ADV),be,verb(V).% only word.pro

verbgroup(vg(VG1,C,VG2),L)  -->{Y is L+1, Y =< 3 },
    verbgroup(VG1,Y),conj1(C),verbgroup(VG2,Y).
verbgroup(vg(VG1,C,VG2),L)  -->{Y is L+1, Y =< 3 },
    verbgroup(VG1,Y),conj3(C),verbgroup(VG2,Y).
verbgroup(vg(VG,P),L)  -->{Y is L+1, Y =< 3 },
    verbgroup(VG,Y),prep(P).
verbgroup(vg(VG,to),L)  -->{Y is L+1, Y =< 3 },
    verbgroup(VG,Y),to.

verbgroup(vg(VE),_)        --> verbgroupend(VE,0).
verbgroup(vg(have,VE),_)   --> have,verbgroupend(VE,0).
verbgroup(vg(have,N,VE),_) --> have,neg(N),verbgroupend(VE,0).
```

```
verbgroup(vg(has,VE),_)    --> has,verbgroupend(VE,0).
verbgroup(vg(has,N,VE),_)  --> has,neg(N),verbgroupend(VE,0).
verbgroup(vg(had,VE),_)    --> had,verbgroupend(VE,0).
verbgroup(vg(had,N,VE),_)  --> had,neg(N),verbgroupend(VE,0).

verbgroup(vg(do,VE),_)     --> do,verbgroupend(VE,0).
verbgroup(vg(do,N,VE),_)   --> do,neg(N),verbgroupend(VE,0).
verbgroup(vg(does,VE),_)   --> does,verbgroupend(VE,0).
verbgroup(vg(does,N,VE),_) --> does,neg(N),verbgroupend(VE,0).
verbgroup(vg(did,VE),_)    --> did,verbgroupend(VE,0).
verbgroup(vg(did,N,VE),_)  --> did,neg(N),verbgroupend(VE,0).

verbgroup(vg(MV,VE),_)   -->
    mv(MV),verbgroupend(VE,0).% only word.pro
verbgroup(vg(MV,N,VE),_)   -->
    mv(MV),neg(N),verbgroupend(VE,0).% only word.pro
verbgroup(vg(MV,have,VE),_)  -->
    mv(MV),have,verbgroupend(VE,0).% only word.pro
verbgroup(vg(MV,N,have,VE),_)-->
    mv(MV),neg(N),have,verbgroupend(VE,0).% only word.pro

%is,are,was,were,
verbgroup(vg(VB,G),_)       --> vb(VB),gerund(G).
verbgroup(vg(VB,N),_)       --> vb(VB),neg(N).
verbgroup(vg(VB,N,G),_)     --> vb(VB),neg(N),gerund(G).
verbgroup(vg(VB,N,V),_)     --> vb(VB),neg(N),verb(V).
verbgroup(vg(VB,ADV),_)     --> vb(VB),adv(ADV).
verbgroup(vg(VB,ADV,G),_)   --> vb(VB),adv(ADV),gerund(G).
verbgroup(vg(VB,ADV,V),_)   --> vb(VB),adv(ADV),verb(V).
verbgroup(vg(VB,N,ADV),_)   --> vb(VB),neg(N),adv(ADV).
verbgroup(vg(VB,N,ADV,G),_) --> vb(VB),neg(N),adv(ADV),gerund(G).
verbgroup(vg(VB,N,ADV,V),_) --> vb(VB),neg(N),adv(ADV),verb(V).

adjectivephrase(ap(AP1,C,AP2),L)  --> {Y is L+1, Y =< 3 },
    adjectivephrase(AP1,Y),conj1(C),adjectivephrase(AP2,Y).
adjectivephrase(ap(AP1,C,AP2),L)  --> {Y is L+1, Y =< 3 },
    adjectivephrase(AP1,Y),conj3(C),adjectivephrase(AP2,Y).
adjectivephrase(ap(ADJ),_)  --> adj(ADJ).

/*objects, subjects, relative clauses*/

nounphrase(np(NP,RC),L,Depth) --> {Y is L+1, Y =< Depth },
    nounphrase(NP,Y,Depth), relativeclause(RC,L,Depth).% only word.pro

%negation
nounphrase(np(no,NP),L,Depth) --> {Y is L+1, Y =< Depth },
    no,nounphrase(NP,Y,Depth).

nounphrase(np(NP1,C,NP2),L,Depth) --> {Y is L+1, Y =< Depth },
```

```
    nounphrase(NP1,Y,Depth),conj1(C),nounphrase(NP2,Y,Depth).
nounphrase(np(NP,PP),L,Depth)  --> {Y is L+1, Y =< Depth },
    nounphrase(NP,Y,Depth),prepphrase(PP,Y,Depth).

nounphrase(np(N),_,_)  --> noun(N).
nounphrase(np(Q,N),_,_) --> quantA(Q),noun(N).
nounphrase(np(Q,N),_,_) --> quantE(Q),noun(N).
nounphrase(np(PRO),_,_) --> pron(PRO).
nounphrase(np(A,N),_,_) --> art(A),noun(N).
nounphrase(np(ADJP,N),_,_)   -->
    adjectivephrase(ADJP,0),noun(N).
nounphrase(np(Q,ADJP,N),_,_) -->
    quantA(Q),adjectivephrase(ADJP,0),noun(N).
nounphrase(np(Q,ADJP,N),_,_) -->
    quantE(Q),adjectivephrase(ADJP,0),noun(N).
nounphrase(np(A,ADJP,N),_,_) -->
    art(A),adjectivephrase(ADJP,0),noun(N).

prepphrase(pp(PRE,NP),L,Depth) -->
    prep(PRE),nounphrase(NP,L,Depth).
prepphrase(pp(to,NP),L,Depth) --> to,nounphrase(NP,L,Depth).

adverbphrase(ap(AP1,AP2),L,Depth) --> {Y is L+1, Y =< Depth },
    adverbphrase(AP1,Y,Depth),adverbphrase(AP2,Y,Depth).
adverbphrase(ap(PP),L,Depth)  --> prepphrase(PP,L,Depth).
adverbphrase(ap(ADV),_,_)  --> adv(ADV).

%objects
objectphrase(op(OBJ),L,Depth)  --> object(OBJ,L,Depth).
objectphrase(op(OBJ1,OBJ2),L,Depth) -->
    object(OBJ1,L,Depth),object(OBJ2,L,Depth).
objectphrase(op(OP1,C,OP2),L,Depth) --> {Y is L+1, Y =< Depth },
    objectphrase(OP1,Y,Depth),conj1(C),objectphrase(OP2,Y,Depth).

object(obj(NP),L,Depth) --> nounphrase(NP,L,Depth).

%verbphrase
verbphrase(vp(VG,RC),L,Depth)     --> {Y is L+1, Y =< Depth },
    verbgroup(VG,0), relativeclause(RC,Y,Depth).% only word.pro
verbphrase(vp(VP1,C,VP2),L,Depth) --> {Y is L+1, Y =< Depth },
    verbphrase(VP1,Y,Depth),conj1(C),verbphrase(VP2,Y,Depth).
verbphrase(vp(VP1,C,VP2),L,Depth) --> {Y is L+1, Y =< Depth },
    verbphrase(VP1,Y,Depth),conj3(C),verbphrase(VP2,Y,Depth).

verbphrase(vp(VG),_,_)            --> verbgroup(VG,0).
verbphrase(vp(VG,OP),L,Depth)     --> {Y is L+1, Y =< Depth },
    verbgroup(VG,0), objectphrase(OP,Y,Depth).
```

```
/* relative clauses */
relativeclause(rc(RC1,C,RC2),L,Depth) --> {Y is L+1, Y =< Depth },
    relativeclause(RC1,Y,Depth),conj1(C),relativeclause(RC2,Y,Depth).
relativeclause(rc(R,VP),L,Depth)      --> {Y is L+1, Y =< Depth },
    rpn1(R), verbphrase(VP,Y,Depth).
relativeclause(rc(R,S),L,Depth)       --> {Y is L+1, Y =< Depth },
    rpn1(R), sentence(S,Y,Depth).
relativeclause(rc(R,S),L,Depth)       --> {Y is L+1, Y =< Depth },
    rpn2(R), sentence(S,Y,Depth).
% what to do.
relativeclause(rc(R,to,V),_,_)             --> rpn2(R), to, verb(V).
relativeclause(rc(comma,S),L,Depth)        --> {Y is L+1, Y =< Depth },
    comma, sentence(S,Y,Depth).
relativeclause(rc(comma,S,comma),L,Depth)  --> {Y is L+1, Y =< Depth },
    comma, sentence(S,Y,Depth),comma.

%sentences
% ... and therefore ...
sentence(s(S1,and,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1 },
    sentence(S1,Y,Depth),and, cond2(C), sentence(S2,Y,Depth).

% and, or, ,
sentence(s(S1,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1 },
    sentence(S1,Y,Depth), conj1(C), sentence(S2,Y,Depth).

% although
sentence(s(S1,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1 },
    sentence(S1,Y,Depth), conj2(C), sentence(S2,Y,Depth).% only word.pro

% but, however
sentence(s(S1,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    sentence(S1,Y,Depth), conj3(C), sentence(S2,Y,Depth).% only word.pro

% therefore
sentence(s(S1,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    sentence(S1,Y,Depth), cond2(C), sentence(S2,Y,Depth).% only word.pro

% if, as,..
sentence(s(S1,C,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    sentence(S1,Y,Depth), cond1(C), sentence(S2,Y,Depth).
sentence(s(C,S1,then,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    cond1(C),sentence(S1,Y,Depth), then, sentence(S2,Y,Depth).% only word.pro
sentence(s(C,S1,comma,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    cond1(C),sentence(S1,Y,Depth), comma, sentence(S2,Y,Depth).
sentence(s(C,S1,then,S2,else,S3),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    cond1(C),sentence(S1,Y,Depth), then, sentence(S2,Y,Depth),
    else,sentence(S3,Y,Depth).% only word.pro
sentence(s(C,S1,then,S2,or,else,S3),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    cond1(C),sentence(S1,Y,Depth), then, sentence(S2,Y,Depth), or,
```

```
    else,sentence(S3,Y,Depth).% only word.pro

%either .. or
sentence(s(either,S1,or,S2),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    either,sentence(S1,Y,Depth),or, sentence(S2,Y,Depth).


/* standard sentence*/
sentence(s(NP,VP),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    nounphrase(NP,Y,Depth),verbphrase(VP,Y,Depth).
sentence(s(AP,NP,VP),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    adverbphrase(AP,Y,Depth),nounphrase(NP,Y,Depth),verbphrase(VP,Y,Depth).
sentence(s(NP,VP,AP),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    nounphrase(NP,Y,Depth),verbphrase(VP,Y,Depth),adverbphrase(AP,Y,Depth).
sentence(s(NP,VP,ADJP),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    nounphrase(NP,Y,Depth),verbphrase(VP,Y,Depth),adjectivephrase(ADJP,0).
sentence(s(NP,VP,ADJP,AP),L,Depth) --> {Y is L+1, Y =< Depth - 1},
    nounphrase(NP,Y,Depth),verbphrase(VP,Y,Depth),adjectivephrase(ADJP,0),
    adverbphrase(AP,Y,Depth).



translate(Sentencelist,Depth) :-
              sentence(Y,0,Depth,Sentencelist,[]),
              !,
              print_tree(Y),
              print_sentence(Y).

translate(_,_) :- halt.

multitranslate(Sentencelist,Depth) :-
              sentence(Y,0,Depth,Sentencelist,[]),
              print_tree(Y),
              print_sentence(Y), fail.

multitranslate(_,_) :- halt.

asktranslate(Sentencelist,Depth) :-
              sentence(Y,0,Depth,Sentencelist,[]),
              print_tree(Y),
              nl,
              write('Is the result of the parsing Ok for you (y/n)?'),
              flush,
              get_single_char(A),
              ( (A == 121,write('y'),!,print_sentence(Y),halt);
                (A \= 121,write('n'),fail) ).

asktranslate(_,_) :- halt.
```

```
asktranslate2(Sentencelist,Depth) :-
              sentence(Y,0,Depth,Sentencelist,[]),
              print_tree(Y),
              nl,
              write('Is the result of the parsing Ok for you (y/n)?'),
              get0(A),skip(10),
              ( (A == 121,
                 print_sentence(Y),
                 nl,
                 write('Do you want to continue the parsing (y/n)?'),
                 get0(B),skip(10),
                  ( (B == 121,fail);
                    (B \= 121,halt) )
                );
                (A \= 121,fail) ).

asktranslate2(_,_) :- halt.
```

B.6 tree.pro

```
ana(Tree,[k(Tree,Pos)],L_outside,R_outside) :-
                          atomic(Tree),
                          !,
                          atomic_length(Tree,N),
                          R_outside is L_outside + N + 2,
                          Pos is (R_outside + L_outside) // 2.

ana(Tree,[k(F,Pos),[k(Successor,Pos)]],L_outside,R_outside):-
            Tree =.. [F,Successor],
            atomic(Successor),
            !,
            atomic_length(Successor,N1),
            atomic_length(F,N2),
            max(N1,N2,N),
            R_outside is L_outside + N + 2,
            Pos is (R_outside + L_outside) // 2.

ana(Tree,[k(F,Pos),List_suc],L_outside,R_outside):-
Tree =.. [F|Successor],
   atomic_length(F,N),
   R_node is L_outside + N +2,
ana_successor(Successor,L_outside,R_sub_tree,List_suc),
calculate_pos(L_outside,Pos,List_suc,R_sub_tree,R_node,R_outside).

ana_successor([Tree|Rest],L_outside,R_outside,L):-
            Rest \= [],
            ana(Tree,L1,L_outside,Mid),
            ana_successor(Rest,Mid,R_outside,L2),
            append(L1,L2,L).

ana_successor([Tree],L_outside,R_outside,L):-
            ana(Tree,L,L_outside,R_outside).

calculate_pos(_,Pos,L,R,M,R):-
            M =< R,
            !,
            first_node(L,Pos1),
            last_node(L,Pos2),
            Pos is (Pos1 + Pos2) // 2.

calculate_pos(Left,Pos,_,R,M,M):-
            M > R,
            Pos is (M + Left) // 2.

print_tree(S) :- nl,
                 ana(S,L,0,_),
                 pr_tree(L).
```

```
pr_tree(L):-
          L = [_|_],
          print_node(L,0),
          nl,
          print_branch(L,0),
          nl,
          print_twig(L,L1,0),
          nl,
          pr_tree(L1).

pr_tree([]).

print_node([X|R],Column1):-
            pr_node(X,Column1,Column2),
            !,
            print_node(R,Column2).

print_node([_|R],Column):-
            !,
            print_node(R,Column).

print_node([],_).

pr_node(k(X,Pos),S1,S2):-
            atomic_length(X,N),
            tab(Pos - N//2 - S1),
            S2 is Pos + N//2 + N mod 2,
            write(X).

/*Leaves*/
print_branch([_,Y|Rest],S):-
            Y \= [_|_],
            !,
            print_branch([Y|Rest],S).

print_branch([_],_) :- !.


/* not-branched, e.g. lexical categories */

print_branch([k(_,Pos),L|Rest],S1):-
            node_num(L,1),
            !,
            tab(Pos-S1),
            write(|),
            S2 is Pos + 1,
            print_branch(Rest,S2).
```

```
/* normal categories */

%-------------------------

print_branch([_,L|Rest],S1) :-
            (\+ (node_num(L,1))),
            !,
            pr_branch(L,S1,S2),
            print_branch(Rest,S2).

print_branch([],_).

/* pr_branch(dominating_node,(Child_left,L1,Child_right,L2]) */

pr_branch(L,S1,S2) :-
             first_node(L,Pos1),
             last_node(L,Pos2),
             Mid is (Pos1+Pos2) // 2,
             tab(Pos1-S1+1),
             n_times(Mid-Pos1-1,'_'),
             write(|),
             n_times(Pos2-Mid-2,'_'),
             (S2 is Pos2 - 1).

first_node([k(_,Pos)|_],Pos).

/*There is only a lst node if there s a first node */

last_node([_|R],Pos) :-
             last_node(R,Pos).

/*1. case: Leaf */
last_node([k(_,Pos)],Pos):- !.

/*2. case: dominating knod */
last_node([k(_,Pos),[_|_]],Pos).

/*Leaves*/
print_twig([_,Y|Rest],Rest1,S1):-
             Y \= [_|_],
             !,
             print_twig([Y|Rest],Rest1,S1).

print_twig([X],[],_):-
             X \= [_|_],
             !.

/* not-branched, e.g. lexical categories */
```

```
print_twig([k(_,Pos),X|Rest],L,S1):-
            node_num(X,1),
            !,
            tab(Pos-S1),
            write(|),
            S2 is Pos + 1,
            print_twig(Rest,L1,S2),
            append(X,L1,L).

/* normal categories */

print_twig([X,[K|R]|Rest],L,S1):-
            !,
            pr_twig(X,[K|R],S1,S2),
            print_twig(Rest,L1,S2),
            append([K|R],L1,L).

print_twig([],[],_).

% pr_twig(dominating_node,(Child_left,L1,Child_right,L2])

pr_twig(k(_,_),L,S1,S4) :-
             first_node(L,Pos1),
             tab(Pos1-S1),
             write(/),
             S2 is Pos1 + 1,
             node_between(L,S2,S3),
             last_node(L,Pos2),
             tab(Pos2 - S3 - 1),
             %write('\'), for some Prolog systems
             write('\\'),
             S4 is Pos2.

node_between([k(_,_)|R],S1,S2) :-
             print_twig_node(R,S1,S2).

print_twig_node([k(_,_)],S,S) :- !.

print_twig_node([k(_,_),[_|_]],S,S) :- !.

print_twig_node([k(_,Pos)|R],S1,S2) :-
             !,
             tab(Pos-S1),
             write(|),
             S3 is Pos +1,
             print_twig_node(R,S3,S2).

print_twig_node([_|R],S1,S2) :-
             print_twig_node(R,S1,S2).
```

```
/*Predicates that help */

node_num([k(_,_)|L],1) :- \+ (member_x(k(_,_),L)).

n_times(Arith,A):- X is Arith,
                   n_times_x(X,A).

n_times_x(N,A) :- N > 0,
                  !,
                  write(A),
                  M is N - 1,
                  n_times(M,A).

n_times_x(N,_) :- N =< 0.

max(X,Y,X):- X >= Y, !.
max(_,Y,Y).


atomic_length(A,N):-  (atom(A);number(A)),
                      !,
                      name(A,L),
                      list_length(L,N).

list_length([_|R],N):- list_length(R,M),
                       !,
                       N is M +1.

list_length([],0).

member_x(X,L):- member(X,L), !.
```

endverbatim

B.7 conv2list.pro

```
% transform the Sentencetree into a List
conv(T) :- conv_list(T,X),write(X).

conv_list(Tree,[Tree])  :-
                          atomic(Tree),
                          !.

conv_list(Tree,[F,[Successor]]):-
            Tree =.. [F,Successor],
            atomic(Successor),
            !.

conv_list(Tree,[F,List_suc]):-
            Tree =.. [F|Successor],
            conv_successor(Successor,List_suc).

conv_successor([Tree|Rest],L):-
            Rest \= [],
            conv_list(Tree,L1),
            conv_successor(Rest,L2),
            append(L1,L2,L).

conv_successor([Tree],L):-
            conv_list(Tree,L).

%-----------------------------------------------------
%print the sentece in list format with
%parsing information in normal format

conv_sentence([]) :- !.

conv_sentence(X) :-
                    atomic(X),
                    !,
                    write(X),
                    write(' ').

conv_sentence([Word]) :-
                   atomic(Word),
                   !,
                   write(Word),
                   write(' ').

conv_sentence([F|Successor]) :-
                              atomic(F),
                              Successor = [Phrase|_],
```

```
                              atomic(Phrase),
                              conv_sentence(F),
                              conv_sentence(Successor).

conv_sentence([F|Successor]) :-
                              atomic(F),
                              Successor = [Phrase|Rest],
                              conv_sentence(Phrase),
                              conv_sentence(Rest).

%-------------------------------------------------------------
%With 2 Arguments to get the sentence in a String

conv_sentence([], String) :-
                    !,
                    name(String,[]).


conv_sentence(X,String) :-
                    atomic(X),
                    !,
                    combine(X,' ',String).

conv_sentence([Word],String) :-
                    atomic(Word),
                    !,
                    combine(Word,' ',String).

conv_sentence([F|Successor],String) :-
                    atomic(F),
                    Successor = [Phrase|_],
                    atomic(Phrase),
                    conv_sentence(F,StringF),
                    conv_sentence(Successor,StringSuc),
                    combine(StringF,StringSuc,String).

conv_sentence([F|Successor],String) :-
                    atomic(F),
                    Successor = [Phrase|Rest],
                    conv_sentence(Phrase,StringPhrase),
                    conv_sentence(Rest,StringRest),
                    combine(StringPhrase,StringRest,String).
%-------------------------------------------------------------
combine(StringA, StringB, StringAB) :-
                    name(StringA,HelplistA),
                    name(StringB,HelplistB),
                    append(HelplistA,HelplistB,List),
                    name(StringAB,List).
```

```
%-----------------------------------------------------------------
% StringPar = (String) and delete the space after the sentence
parentesis(String,ParString) :-
                    combine('(',String,Dummy1),
                    name(Dummy1,Dummy2),
                    append(Dummy3,[0x0020],Dummy2),
                    name(Dummy4,Dummy3),
                    combine(Dummy4,')',ParString).

parentesis(String,ParString) :-
                             combine('(',String,Dummy1),
                             combine(Dummy1,')',ParString).


%-----------------------------------------------------------------
% get the no/not out of the Phrase

%not
move_neg(InputString,OutputString) :-
    name(InputString,HelpListInput),
    append(Help1,[0x006e,0x006f,0x0074,0x0020|Help2],HelpListInput),
    append(Help1,Help2,HelpListOutput),
    name(PreOutputString,HelpListOutput),
    parentesis(PreOutputString,PreOutputStringP),
    name(PreOutputStringP,PreOutputListP),
    append([0x002d],PreOutputListP,OutputList),
    name(OutputString1,OutputList),
    move_neg(OutputString1,OutputString).

%no
move_neg(InputString,OutputString) :-
    name(InputString,HelpListInput),
    append(Help1,[0x006e,0x006f,0x0020|Help2],HelpListInput),
    append(Help1,[0x0073,0x006f,0x006d,0x0065,
        0x002f,0x0061,0x0020|Help2],HelpListOutput),
    name(PreOutputString,HelpListOutput),
    parentesis(PreOutputString,PreOutputStringP),
    name(PreOutputStringP,PreOutputListP),
    append([0x002d],PreOutputListP,OutputList),
    name(OutputString1,OutputList),
    move_neg(OutputString1,OutputString).

move_neg(X,X).
%-----------------------------------------------------------------------
find_or(InputString,Found) :-
    name(InputString,HelpListInput),
    ((append(_,[0x0020,0x006f,0x0072,0x0020|_],HelpListInput),
    Found = or);
    Found = and).
```

```
%-----------------------------------------------------------------------
print_sentence(Sentencetree) :-
conv_list(Sentencetree,Sentencelist),
     !,
     pr_sentence(Sentencelist,Result,_),
     !,
     nl,
     write(Result),
     !,
     pr_pred(Sentencelist,ResultPred,0,_,_),
     !,
     nl,
     postprocessing(ResultPred,ResultPred2),
     to_tree(ResultPred2),
     !,
     nl,
     write(ResultPred2).

%is not necessary, but in case of bad input useful

pr_sentence(Sentencelist,Result,_) :-
     atomic(Sentencelist),
     name(Result,Sentencelist),
     !.

% necessary
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == s,
    !,
    Successor = [Sentence|Rest],
    !,
    pr_sentence(Rest,RestResult,_),
    !,
    pr_sentence(Sentence,ThisResult,_),
    parentesis(ThisResult,ThisResultP),
    combine(ThisResultP,RestResult,Result).

%either I go or you go
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == either,
    !,
    Successor = [s,S1,or,s,S2|Rest],
    !,
    pr_sentence(S1,S1Result,_),
    pr_sentence(S2,S2Result,_),
    !,
```

```
    pr_sentence(Rest,RestResult,_),
    parentesis(S1Result,S1ResultP),
    combine(S1ResultP,' XOR ',S1ResultXOr),
    parentesis(S2Result,S2ResultP),
    combine(S1ResultXOr,S2ResultP,ThisResult),
    combine(ThisResult,RestResult,Result).

%Franz AND Peter like the woods
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
     F == np,
     Successor = [[np,NP1,conj1,[Con],np,NP2]|Rest],
     ( (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(NP2,NP2Result),
               find_or(NP2Result,Found),
        ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
        (ConResult = ' & '))
      )
    ),
    !,
    pr_sentence([np,NP1|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,NP2|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%Yesterday Franz AND Peter like the woods
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == ap,
    Successor = [Adverb,np,[np,NP1,conj1,[Con],np,NP2]|Rest],
    ( (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(NP2,NP2Result),
                      find_or(NP2Result,Found),
           ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
           (ConResult = ' & '))
      )
    ),
    !,
    pr_sentence([ap,Adverb,np,NP1|Rest],Result1,InfoNew),
    !,
    pr_sentence([ap,Adverb,np,NP2|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
```

```
    combine(Dummy,Result2P,Result).

%He buys Susan toys AND Yanna food
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,Pred,op,[op,OP1,conj1,[Con],op,OP2]]|Rest],
    ( (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(OP2,NP2Result),
                     find_or(NP2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
      )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,op,OP1]|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,op,OP2]|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
   combine(Dummy,Result2P,Result).

%I love and hate her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[vg,VG1,conj1,
                [Con],vg,VG2]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VG2,VG2Result),
                      find_or(VG2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,VG1|Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,VG2|Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

% due to redundancy of gramatic
%I love and hate her.
```

```
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[vge,[vge,VGE1,conj1,
                [Con],vge,VGE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VGE2,VGE2Result),
                      find_or(VGE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,[vge,VGE1]|
                 Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,[vge,VGE2]|
                 Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%I have seen AND kissed her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[Pred,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,[Pred,vge,VE1]|
                 Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,[Pred,vge,VE2]|
                 Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).
```

```
%I should see AND kiss her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[mv,Pred,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
      )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,[mv,Pred,vge,VE1]|
                 Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,[mv,Pred,vge,VE2]|
                 Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%I should have seen AND kissed her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[mv,Pred,have,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
      )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,[mv,Pred,have,vge,VE1]|
                 Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,[mv,Pred,have,vge,VE2]|
                 Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).
```

```
%I have NOT seen AND kissed her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[Pred,neg,Neg,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
     ),
     !,
     pr_sentence([np,Subj,vp,[vg,[Pred,neg,Neg,vge,VE1]|
                  Rest1]|Rest2],Result1,InfoNew),
     !,
     pr_sentence([np,Subj,vp,[vg,[Pred,neg,Neg,vge,VE2]|
                  Rest1]|Rest2],Result2,InfoNew),
     parentesis(Result1,Result1P),
     parentesis(Result2,Result2P),
     combine(Result1P,ConResult,Dummy),
     combine(Dummy,Result2P,Result).

%I should NOT see AND kiss her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[mv,Pred,neg,Neg,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,[verb,Pred,neg,Neg,vge,VE1]|
                 Rest1]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,[verb,Pred,neg,Neg,vge,VE2]|
                 Rest1]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
```

```
    combine(Dummy,Result2P,Result).

%I should NOT have seen AND kissed her.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,[mv,Pred,neg,Neg,have,vge,[vge,VE1,conj1,
                [Con],vge,VE2]]|Rest1]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
     ),
     !,
     pr_sentence([np,Subj,vp,[vg,[verb,Pred,neg,Neg,have,vge,VE1]|
                 Rest1]|Rest2],Result1,InfoNew),
     !,
     pr_sentence([np,Subj,vp,[vg,[verb,Pred,neg,Neg,have,vge,VE2]|
                 Rest1]|Rest2],Result2,InfoNew),
     parentesis(Result1,Result1P),
     parentesis(Result2,Result2P),
     combine(Result1P,ConResult,Dummy),
     combine(Dummy,Result2P,Result).

%I go home AND eat pasta.

pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vp,VP1,conj1,[Con],vp,VP2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VP2,VP2Result),
                      find_or(VP2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,VP1|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,VP2|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).
```

```
%I go home BUT eat pasta.
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vp,VP1,conj3,[_],vp,VP2]|Rest],
    ConResult = ' & ',
    InfoNew = and,
    !,
    pr_sentence([np,Subj,vp,VP1|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,VP2|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,HelpResult),
    combine(HelpResult,' & ((Events are contrary) |
            (Event is unexpected))',Result).

%I buy mum AND dad flowers.
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,Pred,op,[obj,[np,[np,OBJ1,conj1,
                [Con],np,OBJ2]]|Rest1]]|Rest2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(OBJ2,OBJ2Result),
                      find_or(OBJ2Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,op,[obj,[np,OBJ1]|
                 Rest1]]|Rest2],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,op,[obj,[np,OBJ2]|
                 Rest1]]|Rest2],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).


%I buy Franz cars and toys.
pr_sentence([F|Successor],Result,InfoOld) :-
     atomic(F),
     F == np,
```

```
     Successor = [Subj,vp,[vg,Pred,op,[obj,OBJ1,obj,[np,[np,OBJ2,conj1,
                 [Con],np,OBJ3]]|Rest1]]|Rest2],
     (  (Con == and,  ConResult = ' & ', InfoNew = and);
        (Con == or,   ConResult = ' | ', InfoNew = or);
        (Con == '(,)', conv_sentence(OBJ3,OBJ3Result),
                       find_or(OBJ3Result,Found),
         ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
         (ConResult = ' & '))
        )
     ),
     !,
     pr_sentence([np,Subj,vp,[vg,Pred,op,[obj,OBJ1,obj,[np,OBJ2]|
                  Rest1]]|Rest2],Result1,InfoNew),
     !,
     pr_sentence([np,Subj,vp,[vg,Pred,op,[obj,OBJ1,obj,[np,OBJ3]|
                  Rest1]]|Rest2],Result2,InfoNew),
     parentesis(Result1,Result1P),
     parentesis(Result2,Result2P),
     combine(Result1P,ConResult,Dummy),
     combine(Dummy,Result2P,Result).

% i know who ... and who ...
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,Pred,rc,
                [rc,RC1,conj1,[Con],rc,RC2]]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(RC2,RC2Result),
                      find_or(RC2Result,Found),
       ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
          (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,rc,RC1]|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,rc,RC2]|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

% i know ... who ... and who ...
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [Subj,vp,[vg,Pred,np,[np,NP,rc,
```

```
                [rc,RC1,conj1,[Con],rc,RC2]]]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(RC2,RC2Result),
                      find_or(RC2Result,Found),
       ((((Found == or); (InfoOld == or) ),ConResult = ' | ');
                         (ConResult = ' & '))
       )
    ),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,np,
                [np,NP,rc,RC1]]|Rest],Result1,InfoNew),
    !,
    pr_sentence([np,Subj,vp,[vg,Pred,np,
                [np,NP,rc,RC2]]|Rest],Result2,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%The man who ... was ....
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == np,
    Successor = [[np,Subj,rc,[_,_,vp,Rc]],vp,Vp|Rest],
    ConResult = ' & ',
    !,
    pr_sentence([np,Subj,vp,Rc|Rest],Result1,_),
    !,
    pr_sentence([np,Subj,vp,Vp|Rest],Result2,_),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%conj Sentence
pr_sentence([F|Successor],Result,InfoOld) :-
    atomic(F),
    F == conj1,
    !,
    Successor = [[Con]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(Rest,HelpResult),
                      find_or(HelpResult,Found),
              ((((Found == or) ; (InfoOld == or) ),
                 ConResult = ' | ');
              (ConResult = ' & '))
       )
```

```
    ),
    !,
    pr_sentence(Rest,RestResult,InfoNew),
    combine(ConResult,RestResult,Result).

%although
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == conj2,
    !,
    Successor = [[_]|Rest],
    ConResult = ' & ',
    !,
    pr_sentence(Rest,RestResult,_),
    combine(ConResult,RestResult,HelpResult),
    combine(HelpResult,' & (Event is unexpected)',Result).

%but, however
pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == conj3,
    !,
    Successor = [[_]|Rest],
    ConResult = ' & ',
    !,
    pr_sentence(Rest,RestResult,_),
    combine(ConResult,RestResult,HelpResult),
    combine(HelpResult,' & ((Events are contrary) |
        (Event is unexpected))',Result).


pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == cond2,
    !,
    Successor = [[_]|Rest],
    CondResult = ' -> ',
    pr_sentence(Rest,RestResult,_),
    combine(CondResult,RestResult,Result).

% .. and therefore ...
pr_sentence([F,G|Successor],Result,_) :-
    atomic(F),
    F == and,
    G == cond2,
    !,
    Successor = [[_]|Rest],
    CondResult = ' -> ',
    pr_sentence(Rest,RestResult,_),
```

```
    combine(CondResult,RestResult,Result).



pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == cond1,
    ( Successor = [[if],s,S1,then,s,S2|Rest];
      Successor = [[_],s,S1,comma,s,S2|Rest]),
    !,
    CondResult = ' -> ',
    pr_sentence(S1,S1Result,_),
    pr_sentence(S2,S2Result,_),
    pr_sentence(Rest,RestResult,_),
    parentesis(S1Result,S1ResultP),
    parentesis(S2Result,S2ResultP),
    combine(S1ResultP,CondResult,HRes1),
    combine(HRes1,S2ResultP,HRes2),
    combine(HRes2,RestResult,Result).

pr_sentence([F|Successor],Result,_) :-
    atomic(F),
    F == cond1,
    !,
    Successor = [[_]|Rest],
    CondResult = ' <- ',
    pr_sentence(Rest,RestResult,_),
    combine(CondResult,RestResult,Result).

pr_sentence(X,Result,_) :-
    conv_sentence(X,PreResult),
    !,
    move_neg(PreResult,Result).
```

endverbatim

B.8 conv2pred.pro

```
postprocessing(InString,OutString) :-
    delete_par(InString,Help1),
    change_is_to_equal(Help1,Help2),
    move_const(Help2,OutString).

print_pred(Sentencetree) :-
    conv_list(Sentencetree,Sentencelist),
    !,
    pr_pred(Sentencelist,Result_par,0,_,_),
    !,
    delete_par(Result_par,Result),
    nl,
    write(Result).


pr_pred(Sentencelist,Result,NumIn,NumIn,_) :-
    atomic(Sentencelist),
    name(Result,Sentencelist),
    !.

% necessary
pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == s,
    !,
    Successor = [Sentence|Rest],
    !,
    pr_pred(Sentence,ThisResult,NumIn,NumInc,_),
    !,
    pr_pred(Rest,RestResult,NumInc,NumOut,_),
    !,
    parentesis(ThisResult,ThisResultP),
    combine(ThisResultP,RestResult,Result).

%either I go or you go
pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == either,
    !,
    Successor = [s,S1,or,s,S2|Rest],
    !,
    pr_pred(S1,S1Result,NumIn,NumOut1,_),
    pr_pred(S2,S2Result,NumOut1,NumOut2,_),
    !,
    pr_pred(Rest,RestResult,NumOut2,NumOut,_),
    parentesis(S1Result,S1ResultP),
    combine(S1ResultP,' XOR ',S1ResultXOr),
```

```
    parentesis(S2Result,S2ResultP),
    combine(S1ResultXOr,S2ResultP,ThisResult),
    combine(ThisResult,RestResult,Result).

%conj Sentence
pr_pred([F|Successor],Result,NumIn,NumOut,InfoOld) :-
    atomic(F),
    F == conj1,
    !,
    Successor = [[Con]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(Rest,HelpResult),
                      find_or(HelpResult,Found),
              ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
              (ConResult = ' & '))
        )
    ),
    !,
    pr_pred(Rest,RestResult,NumIn,NumOut,InfoNew),
    combine(ConResult,RestResult,Result).

%although is transformed into and
pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == conj2,
    !,
    Successor = [[_]|Rest],
    ConResult = ' & ',
    !,
    pr_pred(Rest,RestResult,NumIn,NumOut,_),
    combine(ConResult,RestResult,Result).

%but, however are transformed into and
pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == conj3,
    !,
    Successor = [[_]|Rest],
    ConResult = ' & ',
    !,
    pr_pred(Rest,RestResult,NumIn,NumOut,_),
    combine(ConResult,RestResult,Result).


pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == cond2,
    !,
```

```
    Successor = [[_]|Rest],
    CondResult = ' -> ',
    pr_pred(Rest,RestResult,NumIn,NumOut,_),
    combine(CondResult,RestResult,Result).

% .. and therefore ...
pr_pred([F,G|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == and,
    G == cond2,
    !,
    Successor = [[_]|Rest],
    CondResult = ' -> ',
    pr_pred(Rest,RestResult,NumIn,NumOut,_),
    combine(CondResult,RestResult,Result).



pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == cond1,
    ( Successor = [[if],s,S1,then,s,S2|Rest];
      Successor = [[_],s,S1,comma,s,S2|Rest]),
    !,
    CondResult = ' -> ',
    pr_pred(S1,S1Result,NumIn,NumOut1,_),
    pr_pred(S2,S2Result,NumOut1,NumOut2,_),
    pr_pred(Rest,RestResult,NumOut2,NumOut,_),
    parentesis(S1Result,S1ResultP),
    parentesis(S2Result,S2ResultP),
    combine(S1ResultP,CondResult,HRes1),
    combine(HRes1,S2ResultP,HRes2),
    combine(HRes2,RestResult,Result).

pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == cond1,
    !,
    Successor = [[_]|Rest],
    CondResult = ' <- ',
    pr_pred(Rest,RestResult,NumIn,NumOut,_),
    combine(CondResult,RestResult,Result).

%------------------ end of treating sentences ----------
%Franz AND Peter like the woods
pr_pred([F|Successor],Result,NumIn,NumOut,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [[np,NP1,conj1,[Con],np,NP2]|Rest],
```

```
    ( (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(NP2,NP2Result),
                     find_or(NP2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
    ),                                !,
    pr_pred([np,NP1|Rest],Result1,NumIn,NumMid,InfoNew),
    !,
    pr_pred([np,NP2|Rest],Result2,NumMid,NumOut,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).

%Yesterday Franz AND Peter liked the woods
pr_pred([F|Successor],Result,NumIn,NumOut,InfoOld) :-
    atomic(F),
    F == ap,
    Successor = [Adverb,np,[np,NP1,conj1,[Con],np,NP2]|Rest],
    ( (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(NP2,NP2Result),
                     find_or(NP2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
    ),
    !,
    pr_pred([ap,Adverb,np,NP1|Rest],Result1,NumIn,NumMid,InfoNew),
    !,
    pr_pred([ap,Adverb,np,NP2|Rest],Result2,NumMid,NumOut,InfoNew),
    parentesis(Result1,Result1P),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,Result).


%-----------------end of changing sentences

pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == np,
    !,
    Successor = [Subj,vp,Predandobj|Adverbs],
    !,
    transformNP(Subj,SubjResult,NumIn,NumSubOut,Var,0,ParOut1),
    !,
```

```
    transformVP(Predandobj,[],Adverbs,ResultVP,NumSubOut,NumOut,Var,_),
    combine(SubjResult,ResultVP,PreResult),
    close_Par(ParOut1,StrParentesis),
    combine(PreResult,StrParentesis,Result).

pr_pred([F|Successor],Result,NumIn,NumOut,_) :-
    atomic(F),
    F == ap,
    !,
    Successor = [Adverb1,np,Subj,vp,Predandobj|Adverbs],
    !,
    transformNP(Subj,SubjResult,NumIn,NumSubOut,Var,0,ParOut1),
    !,
    transformVP(Predandobj,Adverb1,Adverbs,
                ResultVP,NumSubOut,NumOut,Var,_),
    combine(SubjResult,ResultVP,PreResult),
    close_Par(ParOut1,StrParentesis),
    combine(PreResult,StrParentesis,Result).

transformVP(Predandobj,Adverb1,Adverbs,Result,NumIn,NumOut,Var,InfoOld):-
    Predandobj = [vp,VP1,conj1,[Con],vp,VP2],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VP2,VP2Result),
                      find_or(VP2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    transformVP(VP1,Adverb1,Adverbs,Result1,NumIn,NumMid,Var,InfoNew),
    parentesis(Result1,Result1P),
    !,
    transformVP(VP2,Adverb1,Adverbs,Result2,NumMid,NumOut,Var,InfoNew),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,PreResult),
    parentesis(PreResult,Result).


transformVP(Predandobj,Adverb1,Adverbs,Result,NumIn,NumOut,Var,InfoOld):-
    Predandobj = [TYP,Pred,op,[op,OP1,conj1,[Con],op,OP2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(OP2,OP2Result),
                      find_or(OP2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
```

```
    ),
    !,
    transformVP([TYP,Pred,op,OP1|Rest],Adverb1,Adverbs,Result1,
                NumIn,NumMid,Var,InfoNew),
    parentesis(Result1,Result1P),
    !,
    transformVP([TYP,Pred,op,OP2|Rest],Adverb1,Adverbs,Result2,
                NumMid,NumOut,Var,InfoNew),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,PreResult),
    parentesis(PreResult,Result).

transformVP(Predandobj,Adverb1,Adverbs,Result,NumIn,NumOut,Var,InfoOld) :-
    Predandobj = [TYP,Pred,op,[obj,[np,[np,OBJ1,conj1,[Con],np,OBJ2]]|Rest]],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(OBJ2,OBJ2Result),
                      find_or(OBJ2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    transformVP([TYP,Pred,op,[obj,[np,OBJ1]|Rest]],Adverb1,Adverbs,
                Result1,NumIn,NumMid,Var,InfoNew),
    parentesis(Result1,Result1P),
    !,
    transformVP([TYP,Pred,op,[obj,[np,OBJ2]|Rest]],Adverb1,Adverbs,
                Result2,NumMid,NumOut,Var,InfoNew),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,PreResult),
    parentesis(PreResult,Result).

%Second Object

transformVP(Predandobj,Adverb1,Adverbs,Result,NumIn,NumOut,Var,_) :-
    seperate(Predandobj,Pred,Objects),
    transformOP(Objects,Result,Pred,Adverb1,
                Adverbs,NumIn,NumOut,Var,0).

transformVERB(ObjectResult,Pred,Adverb1,Adverbs,Result,Var,Vars,ParOut) :-
    ((Vars \= '', combine(Var,',',Help1),
                  combine(Help1,Vars,Varstotal));
    (Varstotal = Var)),
    makeVerb(Pred,Adverb1,Adverbs,VerbResult,Varstotal,_),
    combine(ObjectResult,VerbResult,PreResult),
    close_Par(ParOut,StrParentesis),
```

```
    combine(PreResult,StrParentesis,Result).


transformADJ(Phrase,VarP,Result,InfoOld):-
    Phrase = [ap,AP1,conj1,[Con],ap,AP2],
    ( (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(AP2,AP2Result),
                     find_or(AP2Result,Found),
                (((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
      )
    ),        !,
    transformADJ(AP1,VarP,Result1,InfoNew),
    !,
    transformADJ(AP2,VarP,Result2,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,PreResult),
    parentesis(PreResult,Result).

transformADJ(Phrase,VarP,Result,_):-
    Phrase = [adj,[ADJ]],
    to_pred(ADJ,Pred_),
    delete_(Pred_,Pred),
    combine(Pred,VarP,Result).

transformADJ(Np,NpRest,VarP,Result,_):-
    Np=[ap,ADJ|NpRest],
    transformADJ(ADJ,VarP,AdjPred,_),
    combine(' & ',AdjPred,Result).

transformADJ(Np,NpRest,VarP,Result,_):-
    Np=[art,ART,ap,ADJ|Rest],
    NpRest = [art,ART|Rest],
    transformADJ(ADJ,VarP,AdjPred,_),
  combine(' & ',AdjPred,Result).

transformADJ(Np,Np,_,'',_).

transformNP([F|Successor],NPResult,NumIn,NumOut,Var,ParIn,ParOut) :-
    atomic(F),
    F == quantA,
    !,
    Successor = [_|Np],
    NumOut is NumIn +1,
    combine('x',NumOut,Var),
    parentesis(Var,VarP),
    transformADJ(Np,NpRest,VarP,AdjPred,_),
    to_pred(NpRest,Pred_),
```

```
    delete_(Pred_,Pred),
    combine('all ',Var,Quant),
    combine(Quant,' (',Quant2),
    combine(Quant2,Pred,PreResult1),
    combine(PreResult1,VarP,PreResult2),
    combine(PreResult2,AdjPred,PreResult3),
    combine(PreResult3,' -> (',NPResult),
    ParOut is ParIn + 2.

transformNP([F|Successor],NPResult,NumIn,NumOut,Var,ParIn,ParOut) :-
     atomic(F),
     F == no,
     !,
     Successor = [np,Np],
NumOut is NumIn +1,
     combine('x',NumOut,Var),
     parentesis(Var,VarP),
     transformADJ(Np,NpRest,VarP,AdjPred,_),
     to_pred(NpRest,Pred_),
     delete_(Pred_,Pred),
     combine('all ',Var,Quant),
     combine(Quant,' (',Quant2),
     combine(Quant2,Pred,PreResult1),
     combine(PreResult1,VarP,PreResult2),
     combine(PreResult2,AdjPred,PreResult3),
     combine(PreResult3,' -> -(',NPResult),
     ParOut is ParIn + 2.

transformNP([F|Successor],NPResult,NumIn,NumOut,Var,ParIn,ParOut) :-
    atomic(F),
    F == quantE,
    !,
    Successor = [_|Np],
    NumOut is NumIn +1,
    combine('x',NumOut,Var),
    parentesis(Var,VarP),
    transformADJ(Np,NpRest,VarP,AdjPred,_),
    to_pred(NpRest,Pred_),
    delete_(Pred_,Pred),
    combine('exists ',Var,Quant),
    combine(Quant,' (',Quant2),
    combine(Quant2,Pred,PreResult1),
    combine(PreResult1,VarP,PreResult2),
    combine(PreResult2,AdjPred,PreResult3),
    combine(PreResult3,' & ',NPResult),
    ParOut is ParIn + 1.

%for the pronouns to become constantes
transformNP([F|Successor],'',NumIn,NumIn,Pron,ParIn,ParIn) :-
```

```
    atomic(F),
    F == pron,
    !,
    Successor = [[Pron]].

%for the propnames to become constantes
transformNP([F|Successor],'',NumIn,NumIn,Const,ParIn,ParIn) :-
    atomic(F),
    F == noun,
    Successor = [[propname,[Const]]].


transformNP([F|Successor],NPResult,NumIn,NumOut,Var,ParIn,ParOut) :-
    atomic(F),
    F \= np,
    !,
    NumOut is NumIn +1,
    combine('x',NumOut,Var),
    parentesis(Var,VarP),
    transformADJ([F|Successor],NpRest,VarP,AdjPred,_),
    to_pred(NpRest,Pred_),
    delete_(Pred_,Pred),
    combine('exists ',Var,Quant),
    combine(Quant,' (',Quant2),
    combine(Quant2,Pred,PreResult1),
    combine(PreResult1,VarP,PreResult2),
    combine(PreResult2,AdjPred,PreResult3),
    combine(PreResult3,' & ',NPResult),
    ParOut is ParIn + 1.

transformNP(List,Result,NumIn,NumOut,Var,ParIn,ParOut) :-
    List = [np,Subj,pp,Prep],
    transformNP(Subj,SubResult,NumIn,NumOut,Var,ParIn,ParOut),
    to_pred(Prep,Prep_),
    delete_(Prep_,Pred),
    parentesis(Var,VarP),
    combine(Pred,VarP,PrepResult),
    add_pp(SubResult,PrepResult,Result).

transformNP(List,Result,NumIn,NumOut,Var,ParIn,ParOut) :-
    List = [np,Subj,rc,Prep],
    transformNP(Subj,SubResult,NumIn,NumOut,Var,ParIn,ParOut),
    to_pred(Prep,Prep_),
    delete_(Prep_,Pred),
    parentesis(Var,VarP),
    combine(Pred,VarP,PrepResult),
    add_pp(SubResult,PrepResult,Result).

add_pp(Sub,Prep,Result) :-
```

```
    name(Sub,SubList),
    append(Help1,[0x002d,0x003e,0x0020,0x0028],SubList), %-> (
    append(Help1,[0x0026,0x0020],Help2), %&
    name(Prep,PrepList),
    append(Help2,PrepList,Help3),
    append(Help3,[0x0020,0x002d,0x003e,0x0020,0x0028],PreResult),
    name(Result,PreResult).

add_pp(Sub,Prep,Result) :-
    name(Sub,SubList),
    append(Help1,[0x002d,0x003e,0x0020,0x002d,0x0028],SubList), %-> -(
    append(Help1,[0x0026,0x0020],Help2), %&
    name(Prep,PrepList),
    append(Help2,PrepList,Help3),
    append(Help3,[0x0020,0x002d,0x003e,0x0020,0x002d,0x0028],PreResult),
    name(Result,PreResult).

add_pp(Sub,Prep,Result) :-
    combine(Sub,Prep,Help),
    combine(Help,' & ',Result).

seperate([_,V,op,OP],V,OP) :- !.

seperate(V,V,[]).

% love and hate
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
   Verb = [TYP,[TYP,VG1,conj1,[Con],TYP,VG2]|Rest],
   (  (Con == and,  ConResult = ' & ', InfoNew = and);
      (Con == or,   ConResult = ' | ', InfoNew = or);
      (Con == '(,)', conv_sentence(VG2,VG2Result),
                     find_or(VG2Result,Found),
               ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
               (ConResult = ' & '))
       )
    ),                !,
    makeVerb([TYP,VG1|Rest],Adverb1,Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([TYP,VG2|Rest],Adverb1,Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

% love and hate
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [TYP,VG1,conj1,[Con],TYP,VG2|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VG2,VG2Result),
                      find_or(VG2Result,Found),
```

```
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
    ),
    !,
    makeVerb([TYP,VG1|Rest],Adverb1,Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([TYP,VG2|Rest],Adverb1,Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

%have seen AND kissed
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [Pred,vge,[vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    makeVerb([Pred,vge,VE1|Rest],Adverb1,Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([Pred,vge,VE2|Rest],Adverb1,Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

%should see AND kiss
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [mv,Pred,vge,[vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    makeVerb([mv,Pred,vge,VE1|Rest],Adverb1,Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([mv,Pred,vge,VE2|Rest],Adverb1,Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

%should have seen AND kissed
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
```

```
    Verb = [mv,Pred,have,vge,[vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
      )
     ),
     !,
     makeVerb([mv,Pred,have,vge,VE1|Rest],Adverb1,
              Adverb2,Result1,Vars,InfoNew),
     !,
     makeVerb([mv,Pred,have,vge,VE2|Rest],Adverb1,
              Adverb2,Result2,Vars,InfoNew),
     combine(Result1,ConResult,Dummy),
     combine(Dummy,Result2,Result).

%have NOT seen AND kissed
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [Pred,neg,Neg,vge,[vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
    ),
    !,
    makeVerb([Pred,neg,Neg,vge,VE1|Rest],Adverb1,
             Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([Pred,neg,Neg,vge,VE2|Rest],Adverb1,
             Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

%should NOT see AND kiss
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [mv,Pred,neg,Neg,vge,[vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
        )
    ),
```

```
    !,
    makeVerb([mv,Pred,neg,Neg,vge,VE1|Rest],Adverb1,
             Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([mv,Pred,neg,Neg,vge,VE2|Rest],Adverb1,
             Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

%should NOT have seen AND kissed
makeVerb(Verb,Adverb1,Adverb2,Result,Vars,InfoOld) :-
    Verb = [mv,Pred,neg,Neg,have,vge,
           [vge,VE1,conj1,[Con],vge,VE2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(VE2,VE2Result),
                      find_or(VE2Result,Found),
                ((((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    makeVerb([mv,Pred,neg,Neg,have,vge,VE1|Rest],
             Adverb1,Adverb2,Result1,Vars,InfoNew),
    !,
    makeVerb([mv,Pred,neg,Neg,have,vge,VE2|Rest],
             Adverb1,Adverb2,Result2,Vars,InfoNew),
    combine(Result1,ConResult,Dummy),
    combine(Dummy,Result2,Result).

makeVerb(Verb,Adverb1,Adverb2,Result,Vars,_) :-
    to_pred(Verb,VerbPred),
    to_pred(Adverb1,AdverbPred1),
    to_pred(Adverb2,AdverbPred2),
    combine(VerbPred,AdverbPred1,PreResult1_),
    combine(PreResult1_,AdverbPred2,PreResult2_),
    parentesis(Vars,VarsP),
    delete_(PreResult2_,PreResult),
    combine(PreResult,VarsP,ResultNeg),
    movePredNeg(ResultNeg,Result).


transformOP([],Result,Pred,Adverb1,Adverbs,NumIn,NumIn,Var,ParIn) :-
     transformVERB('',Pred,Adverb1,Adverbs,Result,Var,'',ParIn).

transformOP([F|Successor],Result,Pred,Adverb1,Adverbs,
            NumIn,NumOut,VarOld,ParIn) :-
    atomic(F),
    F == obj,
```

```
    Successor = [Obj],
    Obj = [np,ObjNP],
    !,
    transformNP(ObjNP,OPResult,NumIn,NumOut,Var,ParIn,ParOut),
    transformVERB(OPResult,Pred,Adverb1,Adverbs,Result,VarOld,Var,ParOut).

transformOP([F|Successor],Result,Pred,Adverb1,
            Adverbs,NumIn,NumOut,Var1,ParIn) :-
    atomic(F),
    F == obj,
    Successor = [Obj1,obj,Obj2],
    Obj1 = [np,Obj1NP],
    !,
    transformNP(Obj1NP,ResultObj1,NumIn,NumOut1,Var2,ParIn,ParMid),
    combine(Var1,',',Help1),
    combine(Help1,Var2,Vars),
    transformOP2(Obj2,ResultObj2,Pred,Adverb1,Adverbs,
                 NumOut1,NumOut,Vars,ParMid,_),
    combine(ResultObj1,ResultObj2,Result).

transformOP2([F|Successor],Result,Pred,Adverb1,Adverbs,
             NumIn,NumOut,Var,ParIn,InfoOld) :-
    atomic(F),
    F == np,
    Successor = [[np,OBJ1,conj1,[Con],np,OBJ2]|Rest],
    (  (Con == and,  ConResult = ' & ', InfoNew = and);
       (Con == or,   ConResult = ' | ', InfoNew = or);
       (Con == '(,)', conv_sentence(OBJ2,OBJ2Result),
                      find_or(OBJ2Result,Found),
                (((Found == or) ; (InfoOld == or) ),  ConResult = ' | ');
                (ConResult = ' & '))
       )
    ),
    !,
    transformOP2([np,OBJ1|Rest],Result1,Pred,Adverb1,Adverbs,
                 NumIn,NumMid,Var,0,InfoNew),
    parentesis(Result1,Result1P),
    !,
    transformOP2([np,OBJ2|Rest],Result2,Pred,Adverb1,Adverbs,
                 NumMid,NumOut,Var,ParIn,InfoNew),
    parentesis(Result2,Result2P),
    combine(Result1P,ConResult,Dummy),
    combine(Dummy,Result2P,PreResult),
    parentesis(PreResult,Result).

transformOP2([F|Successor],Result,Pred,Adverb1,
             Adverbs,NumIn,NumOut,VarOld,ParIn,_):-
    atomic(F),
    F == np,
```

```
    Successor = [Obj],
    !,
    transformNP(Obj,OPResult,NumIn,NumOut,Var,ParIn,ParOut),
    transformVERB(OPResult,Pred,Adverb1,Adverbs,
                  Result,VarOld,Var,ParOut).


to_pred([], String) :-
    !,
    name(String,[]).


to_pred(X,String) :-
    atomic(X),
    !,
    combine(X,'_',String).

to_pred([Word],String) :-
    atomic(Word),
    !,
    combine(Word,'_',String).

to_pred([F|Successor],String) :-
    atomic(F),
    Successor = [Phrase|_],
    atomic(Phrase),
    to_pred(F,StringF),
    to_pred(Successor,StringSuc),
    combine(StringF,StringSuc,String).

to_pred([F|Successor],String) :-
     atomic(F),
     Successor = [Phrase|Rest],
     to_pred(Phrase,StringPhrase),
     to_pred(Rest,StringRest),
   combine(StringPhrase,StringRest,String).

delete_(InString,OutString) :-
    name(InString,Dummy2),
    append(Dummy3,[0x005f],Dummy2),
    name(OutString,Dummy3).

delete_par(InString,OutString) :-
    name(InString,Dummy2),
    append([0x0028],Dummy3,Dummy2),
    append(Dummy4,[0x0029],Dummy3),
    name(OutString,Dummy4).

delete_par(X,X).
```

```
change_is_to_equal(InString,OutString) :-
    name(InString,InList),
    ch_ite(InList,OutList),
    name(OutString,OutList).

ch_ite(InList,OutList) :-
    append(Help1,[0x0020,0x0069,0x0073,0x0028|Help2],InList), % is(
    append(Help1,[0x0020,0x0065,0x0071,0x0075,0x0061,
                  0x006c,0x0028|Help2],MidList), % equal(
    ch_ite(MidList,OutList).

ch_ite(InList,OutList) :-
    append(Help1,[0x0020,0x0061,0x0072,0x0065,
                  0x0028|Help2],InList), % are(
    append(Help1,[0x0020,0x0065,0x0071,0x0075,0x0061,
                  0x006c,0x0028|Help2],MidList), % equal(
    ch_ite(MidList,OutList).

ch_ite(X,X).

move_const(InString,OutString) :-
   name(InString,InList),
   mv_c(InList,OutList),
   name(OutString,OutList).

mv_c(InList,OutList) :-
    append(Start,[0x0065,0x0078,0x0069,0x0073,0x0074,
           0x0073,0x0020,0x0078|Help2],InList), %exists x
    append(VarIndex,[0x0020|Help4],Help2),
    append(Help5,[0x0029,0x0020,0x0026,0x0020,0x0065,0x0071,
           0x0075,0x0061,0x006c,0x0028|Help6],Help4), %) & equal(
    append(Predicate,[0x0028,0x0078|VarIndex],Help5), %(x
    append(Const,[0x002c,0x0078|Help7],Help6), %,x
    append(VarIndex,[0x0029,0x0029|Rest],Help7), %))
    append([0x0028],Const,ConstLB),
    append(ConstLB,[0x0029],ConstB),
    append([0x0028],Predicate2,Predicate),
    append(Predicate2,ConstB,Pred_const),
    append(Start,Pred_const,Dummy1),
    append(Dummy1,Rest,MidList),
    mv_c(MidList,OutList).



mv_c(InList,OutList) :-
    append(Start,[0x0065,0x0078,0x0069,0x0073,0x0074,
           0x0073,0x0020,0x0078|Help2],InList), %exists x
    append(VarIndex,[0x0020|Help4],Help2),
```

```
    append(Help5,[0x0029,0x0020,0x0026,0x0020,0x0065,0x0071,
           0x0075,0x0061,0x006c,0x0028|Help6],Help4), %) & equal(
    append(Predicate,[0x0028,0x0078|VarIndex],Help5), %(x
    append(Const,[0x002c,0x0078|Help7],Help6), %,x
    append(VarIndex,[0x0029|Rest],Help7), %)
    append([0x0028],Const,ConstLB),
    append(ConstLB,[0x0029],ConstB),
    append(Predicate,ConstB,Pred_const),
    append(Start,Pred_const,Dummy1),
    append(Dummy1,Rest,MidList),
    mv_c(MidList,OutList).

mv_c(X,X).

%not
movePredNeg(InputString,OutputString) :-
    name(InputString,HelpListInput),
    append(Help1,[0x006e,0x006f,0x0074,0x005f|Help2],HelpListInput),
    append(Help1,Help2,HelpListOutput),
    name(PreOutputString,HelpListOutput),
    parentesis(PreOutputString,PreOutputStringP),
    name(PreOutputStringP,PreOutputListP),
    append([0x002d],PreOutputListP,OutputList),
    name(OutputString,OutputList).

movePredNeg(X,X).

close_Par(0,'').

close_Par(X,StringOut) :-
        Y is X-1,
        close_Par(Y,StringIn),
        combine(StringIn,')',StringOut).

to_tree(PredResult) :-
        name(PredResult,PredList),
        to_tr(PredList,ResultList),
        name(PredTreeString,ResultList),
        atom_to_term(PredTreeString,PredTree,List),
        checklist(call,List),
        !,
        print_tree(PredTree).

to_tr(LiIn,LiOut) :-
        append(Help1,[0x0065,0x0078,0x0069,0x0073,0x0074,
                      0x0073,0x0020|Help2],LiIn), %exists
        append(Help1,[0x0065,0x0078,0x0069,0x0073,0x0074,
                      0x0073,0x005f|Help2],LiMid), %exists_
        to_tr(LiMid,LiOut).
```

```
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0061,0x006c,0x006c,0x0020|Help2],LiIn), %all
        append(Help1,[0x0061,0x006c,0x006c,0x005f|Help2],LiMid), %all_
        to_tr(LiMid,LiOut).

to_tr(LiIn,LiOut) :-
        append(Help1,[0x0020,0x0028|Help2],LiIn), % (
        append(Help1,[0x0028|Help2],LiMid), %(
        to_tr(LiMid,LiOut).

to_tr(LiIn,LiOut) :-
        append(Help1,[0x007c|Help2],LiIn), %|
        append(Help1,[0x006f,0x0072|Help2],LiMid), %or
        to_tr(LiMid,LiOut).

to_tr(LiIn,LiOut) :-
        append(Help1,[0x0020|Help2],LiIn), %' '
        append(Help1,[0x002c|Help2],LiMid), %,
        to_tr(LiMid,LiOut).

%A-Z in a-z
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0041|Help2],LiIn), %A
        append(Help1,[0x0061|Help2],LiMid), %a
        to_tr(LiMid,LiOut).

to_tr(LiIn,LiOut) :-
        append(Help1,[0x0042|Help2],LiIn),
        append(Help1,[0x0062|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0043|Help2],LiIn),
        append(Help1,[0x0063|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0044|Help2],LiIn),
        append(Help1,[0x0064|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0045|Help2],LiIn),
        append(Help1,[0x0065|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0046|Help2],LiIn),
        append(Help1,[0x0066|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0047|Help2],LiIn),
```

```
        append(Help1,[0x0067|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0048|Help2],LiIn),
        append(Help1,[0x0068|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0049|Help2],LiIn),
        append(Help1,[0x0069|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004a|Help2],LiIn),
        append(Help1,[0x006a|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004b|Help2],LiIn),
        append(Help1,[0x006b|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004c|Help2],LiIn),
        append(Help1,[0x006c|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004d|Help2],LiIn),
        append(Help1,[0x006d|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004e|Help2],LiIn),
        append(Help1,[0x006e|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x004f|Help2],LiIn),
        append(Help1,[0x006f|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0050|Help2],LiIn),
        append(Help1,[0x0070|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0051|Help2],LiIn),
        append(Help1,[0x0071|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0052|Help2],LiIn),
        append(Help1,[0x0072|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0053|Help2],LiIn),
        append(Help1,[0x0073|Help2],LiMid),
```

```
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0054|Help2],LiIn),
        append(Help1,[0x0074|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0055|Help2],LiIn),
        append(Help1,[0x0075|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0056|Help2],LiIn),
        append(Help1,[0x0076|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0057|Help2],LiIn),
        append(Help1,[0x0077|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0058|Help2],LiIn),
        append(Help1,[0x0078|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x0059|Help2],LiIn),
        append(Help1,[0x0079|Help2],LiMid),
        to_tr(LiMid,LiOut).
to_tr(LiIn,LiOut) :-
        append(Help1,[0x005a|Help2],LiIn),
        append(Help1,[0x007a|Help2],LiMid),
        to_tr(LiMid,LiOut).

to_tr(X,X).
```

endverbatim

Bibliography

[1] Robert Joan. *Introducció a la lógica.* Centre de publicacions del Campus Nord, UPC Barcelona, 1999.

[2] Rainer Dietrich and Wolfgang Klein. *Computerlinguistik - eine Einführung.* Verlag Kohlhammer, Stuttgard, 1974.

[3] Ch. Lehner. *Prolog und Linguistik.* R. Oldenburg Verlag, München, 1990.

[4] James Allen. *Natural Language Understanding.* The Benjamin/Cumming Publishing Company, California, 1987.

[5] Stuart Russel, Peter Norvig. *Artificial Intelligence: a modern approach.* Prentice-Hall, New Jersey, 1995.

[6] David Till. *Teach Yourself PERl in 21 days.* Sams Publishing, Indianapolis, 1995.

[7] Herbert G. Bohnert, Paul O. Backer. *Automatic Englisch-to-logic translation in a simplified model.* IBM Watson Research Center, 1967.

[8] Patrick Balckburn, Johan Bos. *Representation and Interference for Natural Language. Volume I:Working with First-Order Logic.* `http://www.coli.uni-sb.de/~bos/comsem/book1.html`, 1999.

[9] Patrick Balckburn, Johan Bos. *Representation and Interference for Natural Language. Volume II:Working with Discourse Representation Structures.* `http://www.coli.uni-sb.de/~bos/comsem/book2.html`, 1999.

[10] Peter Hrandek. *Sprache und Logik - Lecture notes.* Universität Wien, Wien, 1999.

[11] D. Gabbay and F. Guenther (eds.). *Handbook of Philosophical Logic, Volume VI.* D. Reidl Publishing Company, 1989.

[12] Walter Kackovsky. *Better Englisch - Grundgrammatik des Englischen* Salzburger Jugend Verlag, Salzburg, 1989.