

Logic-based Question Answering in Dutch

Kirsten Teulen
5908191

Bachelor thesis
Credits: 9 EC

Bachelor Opleiding Kunstmatige Intelligentie

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor

Dr. R. Fernández Rovira

Institute for Logic, Language and Computation
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

2 December, 2011

Abstract

The aim of this this thesis is twofold: To convert the question-answering system by Blackburn and Bos (Curt) [2], which is mainly written in Prolog, to Dutch and to add a new function to the system in the form of the parsing and answering of polar questions. Firstly the English Curt system is discussed. Then the Dutch representations of declarative sentences, wh-questions and polar (yes-or-no) questions are discussed. Finally the answering capabilities of the system are discussed. The final system Dutch Curt is a question-answering system which can do the following things: Parse Dutch declarative sentences, check these sentences for consistency and informativity, parse and answer Dutch wh-questions, and parse and answer Dutch polar questions.

Contents

1	Introduction	4
2	The Curt Family	6
2.1	English Grammar Fragment in Curt	6
2.1.1	Architecture	7
2.1.2	Representation of Wh-Questions	9
2.2	Reasoning in Curt	10
2.2.1	Consistency Check	11
2.2.2	Informativity Check	11
2.2.3	Sample Interaction	11
2.2.4	Background Knowledge	12
2.2.5	Answering Wh-questions	15
3	An Implemented Grammar Fragment for Dutch	17
3.1	Dutch Declarative Sentences	17
3.2	Representation of Dutch Question	19
3.2.1	Representation of Wh-questions	19
3.2.2	Representation of Polar Questions	22
4	Answering Capabilities	25
4.1	Answering Dutch Questions	25
4.1.1	Answering Dutch Wh-questions	25
4.1.2	Answering Dutch Polar Questions	26
4.2	Dutch Background Knowledge	27
4.3	Interacting with Dutch Curt	28
5	Conclusion and Future Work	32
6	Appendices	35
6.1	Installing Dutch Curt	35
6.2	Working with Curt	36

1 Introduction

Computers have become a part of everyday life. They are integrated more and more in our homes and daily schedules. When you want to know something you *google* it or you go to *wikipedia*. How great would it be if you could just ask the question instead of trying to find the right search-terms to find the answer? Or on a smaller scale, what if you had a description of an event and you could ask questions about what is or is not true in that particular event? For the latter case, a closed domain question-answering system would just be the thing.

Question-answering is a broad field with many practical applications, think of chat-bots and other systems a user can query for information. There are roughly two approaches to question-answering: the statistical approach, mostly used for open domain applications [3], and a logic-based approach. The logic-based approach, which is used in this thesis, lies at the intersection of two sub-disciplines of Artificial Intelligence: Natural Language Processing and Logic. Natural language processing is used to parse sentences and questions and to build first-order logic representations for them. These first-order logic representations are then used by several logic-based tools to make the question-answering system work.

There have been several papers about question answering in Dutch, but they tend to focus on the statistical approach. See for example: Ahn et al. (2005) [1].

In their book “Representation and Inference for Natural Language” Blackburn and Bos [2] develop a system in Prolog that initially only parses natural language sentences. Step-by-step they add features until the final product is a question-answering system. This final system, called ‘Helpful Curt’, is a system that can answer who- and what-questions about a small knowledge base. This knowledge base consists of sentences entered by the user and the background knowledge retrieved by Curt. This whole scenario takes places in the Pulp Fiction domain. This Curt system is also used to create a Swedish question-answering system [6].

The aim of this thesis is first to convert the Curt system made by Blackburn and Bos [2] to Dutch, taking in account language differences; second to extend the Curt system with polar questions and the ability to answer those. This will result in a question-answering system which can do the following things:

- Parse Dutch declarative sentences.
- Evaluate these sentences for consistency and informativity.
- Parse and answer Dutch wh-questions.
- Parse and answer Dutch polar questions.

Thesis Outline

In this section the outline of the thesis is discussed by section.

Section 2. In section 2 the original Curt system made by Blackburn and Bos [2] is discussed. The first part, section 2.1, explains the workings of the English grammar fragment and how it's implemented. The second part, section 2.2, deals with the reasoning abilities of Curt introducing several tools to help Curt. Then the background knowledge is discussed and the section ends with discussing the answering of wh-questions.

Section 3. Section 3 details the translation process between the English Curt and the Dutch grammar fragment that was implemented. It starts with section 3.1 on the representation of Dutch declarative sentences. This is followed by section 3.2 on the representation of wh-questions and the added polar questions.

Section 4. Section 4 is about the answering capabilities of the Dutch Curt system. This section is again divided into the answering of Dutch wh-questions in section 4.1.1 and the answering of Dutch polar questions in section 4.1.2. Then the Dutch background knowledge is explained in section 4.2 and everything comes together in the section concerning Dutch Helpful Curt, section 4.3.

Section 5. In section 5 everything is brought together and possible extensions are discussed.

2 The Curt Family

The foundation of the question-answering system of this thesis is the Curt system made by Blackburn and Bos [2]. *Curt* stands for “Clever use of reasoning tools”. The Curt system is implemented in Prolog. It uses a grammar to interpret English sentences and convert them to first-order logic. Curt then uses a model builder and a theorem prover to reason with the first-order logic representation. The Curt system is called the Curt family because it is developed in steps. The first family member is Baby Curt, a very simple and basic modular system on which the other Curts are built. Improving Curt step by step, finally Helpful Curt is developed. Helpful Curt uses Perl scripts to connect the Prolog core of the system with an external theorem prover and an external model builder. The system creates a representation of the discourse and uses these automated reasoning tools to reason with it. Besides building this representation of the discourse, Helpful Curt enables the user to ask certain kinds of questions and is able to answer them.

This section explores Curt, starting with section 2.1 about the English grammar fragment that is used. That section discusses the grammatical architecture of Curt. This is followed by section 2.2 about the reasoning abilities of Curt. A closer look is taken at the consistency and informativity checks as well as the background knowledge. At the end of the section the answering of wh-questions is discussed. Various sections contain examples of Prolog code, this code is used to discuss the structure and the workings of Curt. The precise working of the Prolog predicates and the details of the code are not explained here. For an explanation on these subjects one could best read the textbook by Blackburn and Bos [2].

2.1 English Grammar Fragment in Curt

This section discusses the English grammar fragment used by Blackburn and Bos [2] in the Curt system. The working of this grammar fragment is demonstrated with some examples. The fragment of English can handle several types of words and sentences. These include proper names, nouns, definite and indefinite noun phrases, transitive and intransitive verbs, coordination, negation, relative clauses, and conditional sentences. The verbs are limited to third person singular. The following examples show some sample sentences that Curt can handle, together with the first-order logic semantic representations assigned to them by the system. As can be seen, Prolog predicates are used to represent the logical connectives and the logical quantifiers.

1. Mia dances.
`dance(mia).`
2. Vincent discards a beverage in a hash bar.
`some(A, and(and(some(B, and(hashbar(B), in(A, B))),
beverage(A)), discard(vincent, A))).`
3. The man who dances shrieks and snorts.
`some(A, and(and(dance(A), man(A)), and(shriek(A), snort(A)))).`

4. The woman does not dance.
`some(A, and(woman(A), not(dance(A))))).`
5. If Mia dances Vincent likes a big quarter pounder with cheese.
`imp(dance(mia), some(A, and(and(big(A), qpwc(A)),
like(vincent, A))))).`

2.1.1 Architecture

The grammatical core of Curt consists of four files:

- `englishGrammar.pl` holds the grammatical rules of the portion of English used in Curt.
- `englishLexicon.pl` holds the chosen English lexicon.
- `semLexLambda.pl` contains the semantic representation of the various lexical items described in the lexicon.
- `semRulesLambda.pl` is the counterpart of the grammar and specifies the way the semantic pieces should be put together to form a complete semantic representation of the sentence.

The grammatical core files of Curt render a lambda expression which in turn is processed by the lambda files, which use the lambda calculus to perform semantic composition:

- `lambda.pl`
- `alphaConversion.pl`
- `betaConversion.pl`.

This architecture is visualized in figure 1.

To illustrate this process all the relevant bits of code needed to parse sentence 1 ‘Mia dances’ are explained:

The lexical representations of the proper name (pn) Mia and the intransitive verb (iv) dances are stored in `englishLexicon.pl`.

```
lexEntry(pn, [symbol:mia,syntax:[mia]]).
lexEntry(iv, [symbol:dance,syntax:[dances],inf:fin,num:sg]).
```

The semantic representations of each of these entries are part of the file `semLexLambda.pl`, where the predicate `lam/2` represents the lambda operator.

```
semLex(pn,M):-
  M = [symbol:Sym,
       sem:lam(P,app(P,Sym))].
```

```
semLex(iv,M):-
  M = [symbol:Sym,
       sem:lam(X,Formula)],
  compose(Formula,Sym,[X]).
```

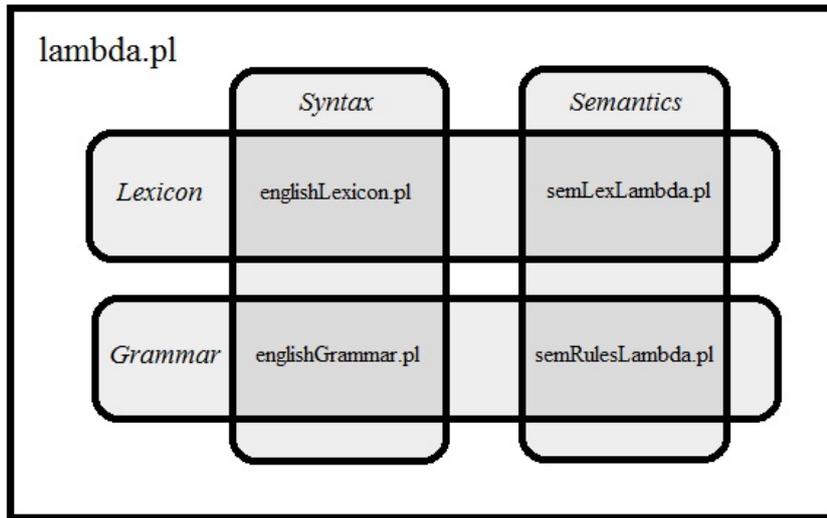


Figure 1: Architecture of the files used by `lambda.pl` [2, p.87]

The following syntactic rules are part of the file `englishGrammar.pl` and are used to parse the sentence.

```
s([coord:no,sem:Sem])-->
  np([coord:_,num:Num,gap:[],sem:NP]),
  vp([coord:_,inf:fin,num:Num,gap:[],sem:VP]),
  {combine(s:Sem,[np:NP,vp:VP])}.

np([coord:no,num:sg,gap:[],sem:NP])-->
  pn([sem:PN]),
  {combine(np:NP,[pn:PN])}.

pn([sem:Sem])-->
  {lexEntry(pn,[symbol:Sym,syntax:Word])},
  Word,
  {semLex(pn,[symbol:Sym,sem:Sem])}.

vp([coord:no,inf:Inf,num:Num,gap:[],sem:VP])-->
  iv([inf:Inf,num:Num,sem:IV]),
  {combine(vp:VP,[iv:IV])}.

iv([inf:Inf,num:Num,sem:Sem])-->
  {lexEntry(iv,[symbol:Sym,syntax:Word,inf:Inf,num:Num])},
  Word,
  {semLex(iv,[symbol:Sym,sem:Sem])}.
```

As a final step in the parsing, the following semantic rules from `semRulesLambda.pl` are used to compose the semantics of each constituent using the predicate `combine/2`.

```
combine(np:NP, [pn:PN]).
combine(vp:A, [iv:A]).
combine(s:app(A,B), [np:A, vp:B]).
```

2.1.2 Representation of Wh-Questions

Wh-questions are questions that ask after a location, a person, an object, the time of an event etc. They start with wh-words like: where, who, what, when. Curt is able to handle who- and what- questions. A user can inquire after persons and objects:

1. Who is a female robber?
`que(A, person(A), some(B, and(and(female(B), robber(B)), eq(A, B))))).`
2. What collapses?
`que(A, thing(A), collapse(A)).`

As can be seen above, the semantic representation of wh-questions includes a question operator `que(Sentence)`. This question operator binds the argument that is being queried, a person or an object. In the code from the file `englishLexicon.pl`, ‘what’ and ‘who’ are declared as wh-words (which Blackburn and Bos call “quantified noun phrases” (`qnp`)). The following code is the lexical entry for ‘what’. Sentence 2 will be used to explain how questions are parsed.

```
lexEntry(qnp, [symbol:thing, syntax:[what], mood:int, type:wh]).
lexEntry(iv, [symbol:collapse, syntax:[collapses], inf:fin, num:sg]).
```

In the `semLexLambda.pl` file important things happen. The representation of the wh-word, differs greatly from that of an ordinary noun phrase. It marks this sentence as a question and makes room for the verb.

```
semLex(qnp, M):-
    M = [type:wh,
         symbol:Sym,
         sem:lam(Q, que(X, Formula, app(Q, X)))],
    compose(Formula, Sym, [X]).
```

```
semLex(iv, M):-
    M = [symbol:Sym,
         sem:lam(X, Formula)],
    compose(Formula, Sym, [X]).
```

The syntactic rules from `englishGrammar.pl` start with a rule specifying that this is indeed a question (`q`). This question must consist of a wh-noun phrase (`whnp`) and a verb phrase (`vp`). The wh-noun phrase in sentence 2 is a wh-word (`qnp`) and the verb phrase an intransitive verb (`iv`).

```

q([sem:Sem])-->
  whnp([num:Num,sem:NP]),
  vp([coord:_,inf:fin,num:Num,gap:[],sem:VP]),
  {combine(q:Sem,[whnp:NP,vp:VP])}.

whnp([num:sg,sem:NP])-->
  qnp([mood:int,sem:QNP]),
  {combine(whnp:NP,[qnp:QNP])}.

qnp([mood:M,sem:NP])-->
  {lexEntry(qnp,[symbol:Symbol,syntax:Word,mood:M,type:Type])},
  Word,
  {semLex(qnp,[type:Type,symbol:Symbol,sem:NP])}.

vp([coord:no,inf:Inf,num:Num,gap:[],sem:VP])-->
  iv([inf:Inf,num:Num,sem:IV]),
  {combine(vp:VP,[iv:IV])}.

iv([inf:Inf,num:Num,sem:Sem])-->
  {lexEntry(iv,[symbol:Sym,syntax:Word,inf:Inf,num:Num])},
  Word,
  {semLex(iv,[symbol:Sym,sem:Sem])}.

```

Finally the right instances of `combine/2` are used from `semRulesLamda.pl`.

```

combine(vp:A,[iv:A]).
combine(whnp:A,[qnp:A]).
combine(q:app(A,B),[whnp:A,vp:B]).

```

2.2 Reasoning in Curt

Curt constructs a representation of the sentences of the discourse (discourse-so-far). When the user enters a new sentence, if it is a declarative sentence and not a question, then the semantic representation of the sentence should be added to the discourse-so-far. But before a sentence is added to the discourse-so-far, two checks are made. First a check is made on the incoming sentence to see if the reading is consistent with the discourse-so-far. Then, if that check is true, a check of informativity is made to see if the sentence is informative with respect to the discourse-so-far. If both of the checks are passed successfully the sentence becomes part of the model that represents the discourse-so-far. To conduct these checks Curt has a number of tools at its disposal. In the remainder of this section these tools are discussed, followed by section 2.2.1 on the consistency check and section 2.2.2 on the informativity check.

- Model checker

A model checker is an automated system that can check whether a first-order-logic formula is true in a first-order-logic model or not. Blackburn and Bos developed their own model checker in Prolog: `modelChecker2.pl` [2].

- Model builder
A model builder is a program that searches for finite models of a formula or set of formulas. Example: Mace4 [4].
- Theorem prover
A theorem prover is a system that provides a systematic method for checking whether or not it is possible to build a model in which a given formula is true or false. Example: Prover9 [5].

2.2.1 Consistency Check

The consistency check works in two directions, a negative approach using a theorem prover and a partial positive test using a model builder.

- Negative test: discourse-so-far $\models \neg\phi$
- Partial positive test: discourse-so-far $\wedge \phi$

If the negation of ϕ logically follows from the discourse-so-far, ϕ must be inconsistent with the discourse-so-far. If a model can be built for the discourse-so-far and ϕ , then ϕ and the discourse-so-far are consistent. It is a partial positive check because it cannot be guaranteed that a model can be found in practical or even real time. That's the reason why these two checks work in tandem with each other and with a time restraint to find an answer.

2.2.2 Informativity Check

The informativity check works in the same dual fashion as the consistency check; Using a theorem prover for the negative test and the model builder for the partial positive test.

- Partial positive test: discourse-so-far $\models \phi$
- Negative test: discourse-so-far $\wedge \neg\phi$

If ϕ follows logically from the discourse-so-far it is not saying anything new, so it is not informative. On the other hand if a model can be built with the discourse-so-far and $\neg\phi$, adding ϕ to the discourse-so-far is eliminating an option and therefore informative.

2.2.3 Sample Interaction

In figure 2 a sample interaction with Curt is shown to illustrate the working of the checks. When Curt is freshly started 'Mia dances' is easily accepted. When a user enters something that negates something that is already in the model of the discourse-so-far the consistency check will make Curt complain. This happens when the blatantly contradictory statement 'Mia does not dance' is entered. When 'Mia dances' is entered again this is of course consistent with the discourse-so-far, which consists only of the fact that 'Mia dances'. That is exactly where the informativity check reacts with 'Well that is obvious' because Curt already knows.

```

?- curt.

> mia dances

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK.

> mia does not dance

Message (consistency checking): prover9 found a result.
Curt: No! I do not believe that!

> mia dances

Message (consistency checking): mace4 found a result.
Message (informativity checking): prover9 found a result.
Curt: Well, that is obvious

```

Figure 2: A sample interaction showing the workings of the consistency and informativity checks in Curt.

2.2.4 Background Knowledge

The reasoning procedure described above, whereby consistency and informativity are checked, is not very useful without knowledge. A user can make Curt believe anything because it has no knowledge of the world. This problem is addressed by the background knowledge. The background knowledge consists of three kinds of knowledge:

- Lexical knowledge

The `lexicalKnowledge.pl` file, the most important knowledge file, holds information about the lexical items and the ontology in which they are ordered. This can be information about the relative position of a noun in the ontology, like ‘a man is a person’ or disjunctions in the ontology like ‘a man is not a woman’. There is also information about the selectional restrictions of verbs, for example: ‘eating is done by either a human or an animal and the thing that is eaten must be edible and food’.

```

lexicalKnowledge(man,1,Axiom):-
    Axiom = all(A,imp(man(A),person(A))).

lexicalKnowledge(woman,1,Axiom):-
    Axiom = all(A,imp(woman(A),not(man(A)))).

lexicalKnowledge(mia,0,Axiom):-
    Axiom = all(A,imp(eq(A,mia),woman(A))).

```

```
lexicalKnowledge(eat,2,Axiom):-
  Axiom = all(X,all(Y,imp(eat(X,Y),
    and(or(person(X),animal(X)),
      and(edible(Y),food(Y))))))).
```

- World knowledge

The `worldKnowledge.pl` file contains knowledge about the functioning of the world, that is, facts like ‘no object can contain itself and an object cannot be part of two distinct objects’.

```
worldKnowledge(have,2,Axiom):-
  Axiom = and(not(some(X,have(X,X))),all(X,all(Y,
    imp(some(Z,and(object(X),and(object(Y),and(object(Z),
      and(have(X,Z),have(Y,Z))))),eq(X,Y)))))).
```

- Situational knowledge

The `situationalKnowledge.pl` file contains information of the situation that is under discussion. In the example described by Blackburn and Bos the situational knowledge is that there are at least two cars. In other situations that might not be true, but those situations will have a different situational knowledge file, whereas the `lexicalKnowledge.pl` and `worldKnowledge.pl` files in principle hold in every situation.

```
Axiom = some(X,some(Y,and(car(X),and(car(Y),not(eq(X,Y)))))).
```

While distinguishing between the above types of background knowledge is a kind of arbitrary distinction, it is a functional one. The background knowledge is used in accord with the discourse-so-far in the consistency and informativity checks described in section 2.2.1 and section 2.2.2. For example, the negative consistency check will look like this:

$$\text{lexicalknowl.} \wedge \text{worldknowl.} \wedge \text{situationalknowl.} \wedge \text{discourse-so-far} \models \neg\phi$$

An example showing one of the ways Curt uses background knowledge is shown in figure 3. The example starts with creating a new, empty discourse. The sentence ‘Mia dances’ causes no trouble. Then ‘Mia is a man’ is entered by the user. In the model it is visible that ‘Mia’ is a woman. These two lexical knowledge facts are used:

```
lexicalKnowledge(mia,0,Axiom):-
  Axiom = all(A,imp(eq(A,mia),woman(A))).
```

```
lexicalKnowledge(woman,1,Axiom):-
  Axiom = all(A,imp(woman(A),not(man(A)))).
```

The first one makes sure that the fact that ‘Mia’ is a woman is in the model. The second one concludes from the fact that ‘Mia is a woman’ that she cannot be a man.

```
> new

> mia dances

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK.

> mia is a man

Message (consistency checking): prover9 found a result.
Curt: No! I do not believe that!

> models.

1 D=[d1, d2]
f(0, mia, d1)
f(1, animal, [])
f(1, dance, [d1])
f(1, entity, [d1])
f(1, event, [])
f(1, man, [])
f(1, object, [])
f(1, organism, [d1])
f(1, person, [d1])
f(1, thing, [d1])
f(1, woman, [d1])
```

Figure 3: A sample interaction showing one of the ways Curt uses background knowledge.

2.2.5 Answering Wh-questions

As described in section 2.1.2 Helpful Curt can handle two kinds of wh-type of questions, who- and what-questions:

1. Who is a female robber?
`que(A, person(A), some(B, and(and(female(B), robber(B)), eq(A, B))))).`
2. What collapses?
`que(A, thing(A), collapse(A)).`

The first step in finding an answer is using a model checker to see if there is a way to satisfy the parsed question in the built-up discourse. However, a model checker cannot handle questions by itself. Before a question can be processed by `satisfy/4`, the main predicate of `modelChecker2.pl`, it needs to be changed to an existential formula. The code for that transformation from `helpfulCurt.pl` is:

```
que(X,R,S),
some(X,and(R,S)).
```

This will transform question 1 and 2 to:

1. Who is a female robber?
`que(A, person(A), some(B, and(and(female(B), robber(B)), eq(A, B))))).`
`some(A, and(person(A), some(B, and(and(female(B), robber(B)), eq(A, B))))).`
2. What collapses?
`que(A, thing(A), collapse(A)).`
`some(A, and(thing(A), collapse(A))).`

If there is no way of satisfying the existential representation of the question, Curt does not know the answer to the question. In that case Curt is finished and will tell the user that it doesn't know the answer. On the other hand, if there is at least one way to satisfy the representation of the question, to actually answer the question more work needs to be done. A `findall` statement is employed to find all the people or things that are an answer to the question. This set of answers is then processed by a predicate called `realiseAnswers` to give a sensible answer to the user instead of merely a set. The model checker used for the wh-questions is `modelChecker2.pl`.

In figure 4 an example of Curt dealing with a wh-question is shown.

```
?- curt.  
  
> mia likes every robber  
  
Message (consistency checking): mace4 found a result.  
Message (informativity checking): mace4 found a result.  
Curt: OK.  
  
> vincent is a robber  
  
Message (consistency checking): mace4 found a result.  
Message (informativity checking): mace4 found a result.  
Curt: OK.  
  
> who does mia like?  
  
Message (answer checking): prover9 found result "proof".  
Curt: This question makes sense!  
  
Curt: vincent
```

Figure 4: A sample interaction showing how Curt deals with a wh-question.

3 An Implemented Grammar Fragment for Dutch

Taking the Curt system described in section 2 as a starting point, a Dutch grammar fragment is implemented. The lexical fragment used by the programs described in sections 3 and 4 focuses on swimming games. The Lexicon is a small one, having over 40 entries including 8 nouns, 5 proper names, 2 wh-words, 3 intransitive verbs, 4 transitive verbs and 5 adjectives. In section 3.1 declarative sentences in Dutch are discussed. That is followed by section 3.2 where the representation of wh-questions and polar questions in Dutch are discussed.

3.1 Dutch Declarative Sentences

Dutch and English are similar languages. Some of the grammatical constructs are the same, but a word by word translation would still yield botched and incomprehensible sentences. Here are a few sample sentence that can be constructed with the `dutchGrammar` and `dutchLexicon` and their first-order logic representations in Prolog:

1. Moniek zwemt.
`zwemmen(moniek)`
2. Een sporter wint.
`some(A, and(sporter(A), winnen(A)))`
3. Moniek Nijhuis praat met Pieter.
`pratenmet(moniekN, pieter)`
4. Moniek en Ranomi feliciteren Pieter.
`and(feliciteren(moniek, pieter), feliciteren(ranomi, pieter))`
5. Pieter is een zwemmer.
`some(A, and(zwemmer(A), eq(pieter, A)))`

These sentences are all similar to their English counterparts. Sentences 1 and 2 show intransitive verbs with a pronoun and full noun phrase, respectively. Sentence 3 shows a transitive verb and a double name `Moniek Nijhuis`, a combination of two words, a first name and a last name, that refer together to one entity. The representation of that entity is `moniekN`. The verb is considered transitive and translated together with its preferred preposition to `pratenmet(X)`. While this is not entirely correct, it allows for easy translation of declarative sentences. This dirty trick will however pose problems with the polar questions, which will be discussed in section 3.2.2. Sentence 4 shows coordination on the noun phrase level as well as a transitive verb. Sentence 5 shows the use of an equality between two noun phrases declaring that the person named Pieter is a swimmer.

Below, the important pieces of code that help construct sentence 5 are shown. Dutch Curt is like the original Curt in the aspect that it has a grammatical core consisting of four files. The `semRulesLambda.pl` and `semLexLambda.pl` files have undergone some changes and additions which will be discussed when encountered. The `englishLexicon.pl` and `englishGrammar.pl` are replaced

with `dutchLexicon.pl` and `dutchGrammar.pl`. In `dutchLexicon.pl` all the lexical entries for sentence 5 are coded:

```
lexEntry(pn, [symbol:pieter, syntax:[pieter]], persoon).
lexEntry(cop, [pol:pos, syntax:[is], inf:fin, num:sg]).
lexEntry(det, [syntax:[een], mood:decl, type:indef]).
lexEntry(noun, [symbol:zwemmer, syntax:[zwemmer]], persoon).
```

The semantic representation for the words is recovered from `semLexLambda.pl`. They are unchanged from the original Curt.

```
semLex(pn, M) :-
    M = [symbol:Sym,
         sem:lam(P, app(P, Sym))].

semLex(cop, M) :-
    M = [pol:pos,
         sem:lam(K, lam(Y, app(K, lam(X, eq(Y, X)))))].

semLex(det, M) :-
    M = [type:indef,
         sem:lam(P, lam(Q, some(X, and(app(P, X), app(Q, X)))))].

semLex(noun, M) :-
    M = [symbol:Sym,
         sem:lam(X, Formula)],
        compose(Formula, Sym, [X]).
```

In the `dutchGrammar.pl` file the code for the syntactic rules is stored. These particular syntactic rules have not changed from the original Curt.

```
s([coord:no, sem:Sem])-->
    np([coord:_, num:Num, gap:[], sem:NP]),
    vp([coord:_, inf:fin, num:Num, gap:[], sem:VP]),
    {combine(s:Sem, [np:NP, vp:VP])}.

np([coord:no, num:sg, gap:[], sem:NP], Type)-->
    pn([sem:PN], Type),
    {combine(np:NP, [pn:PN])}.

pn([sem:Sem], Type)-->
    {lexEntry(pn, [symbol:Sym, syntax:Word], Type)},
    Word,
    {semLex(pn, [symbol:Sym, sem:Sem])}.

vp([coord:no, inf:Inf, num:Num, gap:[], sem:VP])-->
    cop([inf:Inf, num:Num, sem:Cop]),
    np([coord:_, num:_, gap:[], sem:NP]),
    {combine(vp:VP, [cop:Cop, np:NP])}.

vp([coord:no, inf:Inf, num:Num, gap:[], sem:VP], _)-->
    cop([inf:Inf, num:Num, sem:Cop]),
```

```

np([coord:_, num:_, gap:[], sem:NP], _),
{combine(vp:VP, [cop:Cop, np:NP])}.

np([coord:no, num:sg, gap:[], sem:NP], Type)-->
det([mood:decl, type:_, sem:Det]),
n([coord:_, sem:N], Type),
{combine(np:NP, [det:Det, n:N])}.

det([mood:M, type:Type, sem:Det])-->
{lexEntry(det, [syntax:Word, mood:M, type:Type])},
Word,
{semLex(det, [type:Type, sem:Det])}.

noun([sem:Sem], Type)-->
{lexEntry(noun, [symbol:Sym, syntax:Word], Type)},
Word,
{semLex(noun, [symbol:Sym, sem:Sem])}.

```

These semantic representations are then combined using various unchanged instances of `combine/2` from `semRulesLambda.pl`.

```

combine(np:A, [pn:A]).
combine(np:app(A,B), [det:A, n:B]).
combine(vp:app(A,B), [cop:A, np:B]).
combine(s:app(A,B), [np:A, vp:B]).

```

3.2 Representation of Dutch Question

In this section the syntactic and semantic representation of questions in Dutch are discussed. As explained in section 2.2.5, the original English system by Blackburn and Bos only handles two types of wh-questions. In our system wh-questions are implemented in Dutch and the original capabilities of the system have been extended by adding polar questions.

3.2.1 Representation of Wh-questions

Blackburn and Bos introduce two kinds of wh-questions in the Curt system, described in section 2.1.2. Dutch Curt covers the same two types: the who- and the what-questions. In Dutch wh-questions look like this:

1. Wie zwemt?
que(A, persoon(A), zwemmen(A))
2. Wat krijgt Moniek?
que(A, object(A), krijgen(moniek, A))
3. Wie krijgt de medaille?
que(A, persoon(A), some(B, and(medaille(B), krijgen(A, B))))

In Dutch a wh-question always starts with a wh-word followed by a verb. If the verb is intransitive that is the whole question. If a verb is transitive, the resulting syntactic structure depends on whether the question inquires about the subject or the object. If the question inquires about the subject, as in sentences 1 and 3, the subject noun phrase is replaced by the wh-word. If the question inquires about the object the verb, and subject are swapped. This is commonly known as “subject-verb inversion” and the mechanism is quite different from the one used for English wh-questions. In English the place of the swapped verb is taken by an auxiliary verb. Sentence 2 in English would be ‘What does Moniek get?’.

Although the syntactic structure of the wh- questions asking after the subject and the wh-questions asking after the object is different, in Dutch there is no difference between a subject noun or proper name and a object noun or proper name. The result of this is that Curt cannot distinguish between question 2 and 3. A human knows that in question two the word ‘wat’ is inquiring after the object and in sentence 3 ‘wie’ is inquiring after the subject. The reason a human knows this is because the verb ‘krijgen’ has a person-subject and a thing-object. How Curt learn this distinction? The decision was made to include an extra argument in the lexical items, rather than trying to establish a link to the lexical knowledge base, where similar information is stored. This choice was made to keep the structure of Curt intact and to avoid Curt making many extra calls. The lexical entries for sentence 2 are shown below. ‘wat’ is declared a thing, ‘moniek’ is declared a person and in the representation of the verb ‘krijgen’ is specified that it takes a person-subject and a thing-object.

```
lexEntry(qnp, [symbol:object,syntax:[wat],mood:int,type:wh],ding).
lexEntry(tv, [symbol:krijgen,syntax:[krijgt],inf:fin,num:sg],
[subj:persoon,obj:ding]).
lexEntry(pn, [symbol:moniek,syntax:[moniek]],persoon).
```

The representation of a wh-question, in English and Dutch, comes from the wh-words. The wh-words introduce a question operator (`que`). This question operator and the semantic structure that comes with it determine the form of the wh-questions. The working of the wh-word, the subject-verb inversion and the working of the new argument is demonstrated with sentence 2.

Just like English wh-questions, the code for the lexical items are looked up in the lexicon, `dutchLexicon`. ‘wat’ is found to be a wh-word (`qnp`), ‘krijgt’ a transitive verb (`tv`) and ‘moniek’ a proper name (`pn`). Then the semantic representations are extracted from `semLexLambda`. The representation of the wh-word (`qnp`) is still the most important part of the question and unchanged from the original Curt.

```
semLex(qnp,M):-
  M = [type:wh,
       symbol:Sym,
       sem:lam(Q,que(X,Formula,app(Q,X)))],
  compose(Formula,Sym,[X]).
```

```
semLex(tv,M):-
  M = [symbol:Sym,
        sem:lam(K,lam(Y,app(K,lam(X,Formula))))],
  compose(Formula,Sym,[Y,X]).
```

```
semLex(pn,M):-
  M = [symbol:Sym,
        sem:lam(P,app(P,Sym))].
```

The syntactic information is retrieved from `dutchGrammar.pl`. It starts with a question (q) which consist of a wh-noun phrase (`whnp`) and an inverted sentence (`sinv`). The wh-noun phrase in turn consists of a wh-word (`qnp`) and the inverted sentence of a verb phrase (`vp`) and a noun phrase (`np`). This verb phrase is a transitive verb (`tv`) with a gap noun phrase, the noun phrase is a proper name (`pn`).

```
q([sem:Sem])-->
  whnp([num:_,sem:NP],Obj),
  sinv([gap:[np:NP],sem:S],[subj:_,obj:Obj]),
  {combine(q:Sem,[sinv:S])}.
```

```
whnp([num:sg,sem:NP])-->
  qnp([mood:int,sem:QNP]),
  {combine(whnp:NP,[qnp:QNP])}.
```

```
qnp([mood:M,sem:NP])-->
  {lexEntry(qnp,[symbol:Symbol,syntax:Word,mood:M,type:Type])},
  Word,
  {semLex(qnp,[type:Type,symbol:Symbol,sem:NP])}.
```

```
sinv([gap:G,sem:S],[subj:Subj,obj:Obj])-->
  vp([coord:_,inf:_,num:Num,gap:G,sem:VP],[subj:Subj,obj:Obj]),
  np([coord:_,num:Num,gap:[],sem:NP],Subj),
  {combine(sinv:S,[np:NP,vp:VP])}.
```

```
vp([coord:no,inf:I,num:Num,gap:G,sem:VP],[subj:Subj,obj:Obj])-->
  tv([inf:I,num:Num,sem:TV],[subj:Subj,obj:Obj]),
  np([coord:_,num:_,gap:G,sem:NP],Obj),
  {combine(vp:VP,[tv:TV,np:NP])}.
```

```
tv([inf:Inf,num:Num,sem:Sem],Type)-->
  {lexEntry(tv,[symbol:Sym,syntax:Word,inf:Inf,num:Num],Type)},
  Word,
  {semLex(tv,[symbol:Sym,sem:Sem])}.
```

```
np([coord:no,num:sg,gap:[np:NP],sem:NP],_)--> [].
```

```
np([coord:no,num:sg,gap:[],sem:NP],Type)-->
  pn([sem:PN],Type),
  {combine(np:NP,[pn:PN])}.
```

```
pn([sem:Sem],Type)-->
  {lexEntry(pn,[symbol:Sym,syntax:Word],Type)},
  Word,
  {semLex(pn,[symbol:Sym,sem:Sem])}.
```

These semantic representations are then combined using various instances of `combine/2` from the `semRulesLambda.pl` file.

```
combine(np:A,[pn:A]).
combine(vp:app(A,B),[tv:A,np:B]).
combine(sinv:app(A,B),[np:A,vp:B]).
combine(whnp:A,[qnp:A]).
combine(q:A,[sinv:A]).
```

3.2.2 Representation of Polar Questions

Polar questions are questions which only have two possible answers: ‘yes’ or ‘no’. They are clearly distinct from wh-questions which have a definite answer, usually a noun or proper name. In Dutch a polar question stands out because the first word of the question is the verb. In English polar questions begin with an auxiliary verb. The verb is followed by the subject of the verb. If there is an object, it will follow the subject. In other words polar questions have subject-verb inversion, just like the wh-questions that ask after an object, which is explained in section 3.2.1. In the Dutch grammar fragment used, a polar question can have the following basic forms and representation:

1. Zwemt Pieter?
`pq(zwemmen(pieter))`
2. Krijgt Pieter de medaille?
`pq((some(A, and(medaille(A),krijgen(pieter, A)).`

Because there are no question words that signal ‘this is a question’ in polar questions, the difference in representation does not come from the bottom but from the top. The word order is responsible for the fact that a particular utterance is indeed a polar question. The representation is that of a declarative sentence adorned with the polar question operator. The operator signals that it should not be added to the discourse-so-far but that it should be evaluated true or false regarding the discourse-so-far. How a polar question is parsed is explained below using question 2 as an example.

Because there are no special things going on at the word level the lexical entries from `dutchLexicon.pl` are the same as they would be for ‘Pieter krijgt de medaille’.

```
lexEntry(tv,[symbol:krijgen,syntax:[krijgt],inf:fin,num:sg],
  [subj:persoon,obj:ding]).
lexEntry(pn,[symbol:pieter,syntax:[pieter]],persoon).
lexEntry(det,[syntax:[de],mood:decl,type:def]).
lexEntry(noun,[symbol:medaille,syntax:[medaille]],ding).
```

The same goes for the semantic representation on word-level from `semLexLambda.pl`.

```
semLex(tv,M):-
    M = [symbol:Sym,
          sem:lam(K,lam(Y,app(K,lam(X,Formula))))],
    compose(Formula,Sym,[Y,X]).

semLex(det,M):-
    M = [type:def,
          sem:lam(P,lam(Q,some(X,and(app(P,X),app(Q,X)))))].

semLex(noun,M):-
    M = [symbol:Sym,
          sem:lam(X,Formula)],
    compose(Formula,Sym,[X]).

semLex(pn,M):-
    M = [symbol:Sym,
          sem:lam(P,app(P,Sym))].
```

In the code from `dutchGrammar.pl` the polar question structure is clearly visible. A polar question consists of an transitive verb (`tv`) followed by two noun phrases (`np`). The substance of these transitive verb and noun phrases has all the same possibilities as they have in declarative sentences.

```
pq([sem:Sem])-->
    tv([inf:_,num:_,sem:TV],_),
    np([coord:_,num:_,gap:_,sem:NP],_),
    np([coord:_,num:_,gap:_,sem:NP2],_),
    {combine(pq:Sem,[tv:TV,np:NP,np:NP2])}.

tv([inf:Inf,num:Num,sem:Sem],Type)-->
    {lexEntry(tv,[symbol:Sym,syntax:Word,inf:Inf,num:Num],Type)},
    Word,
    {semLex(tv,[symbol:Sym,sem:Sem])}.

np([coord:no,num:sg,gap:[],sem:NP],Type)-->
    pn([sem:PN],Type),
    {combine(np:NP,[pn:PN])}.

np([coord:no,num:sg,gap:[],sem:NP],Type)-->
    det([mood:decl,type:_,sem:Det]),
    n([coord:_,sem:N],Type),
    {combine(np:NP,[det:Det,n:N])}.

det([mood:M,type:Type,sem:Det])-->
    {lexEntry(det,[syntax:Word,mood:M,type:Type])},
    Word,
    {semLex(det,[type:Type,sem:Det])}.
```

```

noun([sem:Sem],Type)-->
  {lexEntry(noun,[symbol:Sym,syntax:Word],Type)},
  Word,
  {semLex(noun,[symbol:Sym,sem:Sem])}.

```

```

pn([sem:Sem],Type)-->
  {lexEntry(pn,[symbol:Sym,syntax:Word],Type)},
  Word,
  {semLex(pn,[symbol:Sym,sem:Sem])}.

```

These semantic representations are then combined using various instances of `combine` from the `semRulesLambda.pl` file. The object noun phrase is integrated in the verb phrase (`vp`) as usual. The subject noun phrase is resolved at the polar question level as an ordinary subject would, although it had a different position in the polar question. Finally at the top level the `pq(Sentence)` operator is added to the sentence. This makes the final representation of a polar question that of a declarative sentence with a polar question operator.

```

combine(np:app(A,B),[det:A,n:B]).
combine(vp:app(A,B),[tv:A,np:B]).
combine(np:A,[pn:A]).
combine(pq:app(C,app(A,B)),[tv:A,np:C,np:B]).
combine(t:pq(Converted),[pq:Sem]).

```

Because of the particular polar questions structure treating verbs with a preferred preposition as transitive, as explained in section 3.1, turns out to be not so clever after all.

1. Praat Pieter met Ranomi?
2. *Praat met Pieter Ranomi?

Question 1 is the correct Dutch form for polar question with a preposition. Question 2 is the question that Curt can parse. When parsing question 2 Curt will give the correct answer to question 1. What happened is that the dirty trick explained in section 3.1 binds the preposition next to the verb. That will not cause problems in declarative sentences and *wh*-questions but it changes a polar question into something that no properly educated Dutch person will ever use. Although this clearly is not an ideal way of dealing with prepositions in Dutch, prepositions are not the focus of this thesis, therefore this representation is used anyway.

4 Answering Capabilities

This section discusses the reasoning and answering capabilities of the Dutch Curt. It starts with section 4.1 about the answering of Dutch questions. It is divided in two, a short part about answering Dutch wh-questions and a larger part about the answering of polar questions in Dutch. That is followed by section 4.2 on the background knowledge used in Dutch Curt. The last section, section 4.3 brings everything together in the form of Dutch Curt. There some sample interactions with Dutch Curt are presented and explained.

4.1 Answering Dutch Questions

Dutch questions, like any questions, demand answers. In section 4.1.1 wh-questions are briefly discussed. In section 4.1.2 polar questions are discussed in more detail.

4.1.1 Answering Dutch Wh-questions

In section 2.2.5 the transformation of wh- questions into existential statements was discussed. Although there are some differences between Dutch and English wh-questions, discussed in section 3.2.1, the same code for the transformation from `helpfulCurt.pl` is applied to Dutch questions. The code:

```
que(X,R,S),
some(X,and(R,S)).
```

Although the syntactic structure is different in English and Dutch, the semantic representation is the same, which is exactly what was wanted:

1. Wie zwemt?
`que(A, persoon(A), zwemmen(A)).`
`some(A, and(persoon(A),zwemmen(A))).`
2. Wat krijgt Moniek?
`que(A, object(A), krijgen(moniek, A)).`
`some(A, and(object(A),krijgen(moniek, A))).`
3. Wie krijgt de medaille?
`que(A, persoon(A), some(B, and(medaille(B), krijgen(A, B)))).`
`some(A, and(persoon(A), some(B, and(medaille(B), krijgen(A, B)))).`

Because the semantic representation for Dutch wh-questions is the same as for English wh-questions, answering them proceeds in the same way: the model checker is called and a `findall` is used to retrieve answers to the questions. The relevant lines of code lifted out from the whole `answerQuestion` predicate:

```
answerQuestion(que(X,R,S),Models,Moves):-
    satisfy(some(X,and(R,S)),Model,[],Result),
    findall(A,satisfy(and(R,S),Model,[g(X,A)],pos),Answers),
    realiseAnswer(Answers,que(X,R,S),Model,String),
```

To make the Prolog code easier to understand, the calls that would be made to answer question 1 are shown below:

```
answerQuestion(que(X, persoon(X), zwemmen(X)),Models,Moves):-
    satisfy(some(X,and(persoon(X),zwemmen(X))),Model,[],Result),
    findall(A,satisfy(and(persoon(X),zwemmen(X)),Model,[g(X,A)],
    pos),Answers),
    realiseAnswer(Answers,que(X, persoon(X), zwemmen(X)),Model,
    String),
```

The first call to `satisfy/4` is made to see if there is any way to satisfy the existential statement that is converted from question . Then the `findall` is employed to find all *A* that can be assigned to the place of *X* in the model. All the different ways to bind variable *X* are then stored in a list called *Answers*. This list with bound variables is then processed in `realiseAnswers` together with the original form of question 1 to procure an answer in a user friendly form.

4.1.2 Answering Dutch Polar Questions

In section 3.2.2 the representation of polar questions is discussed. In contrast to wh-questions, polar questions, like the questions below, require a yes-or-no answer. To procure this yes-or-no answer, a query to the discourse-so-far needs to be made.

1. Zwemt Pieter?
pq(zwemmen(pieter))
2. Krijgt Pieter de medaille?
pq((some(A, and(medaille(A),krijgen(pieter, A)).

The declarative sentence adorned with the polar question operator is sent to the `answerQuestion` predicate, where it matches with the predicate for the polar questions:

```
answerQuestion(pq(Sentence),Models,Moves):-
    (
        Models=[Model|_],
        satisfy(Sentence,Model,[],Result),
        \+ Result=undef,
        !,
        Moves=[sensible_question,answerpq(Result)]
    );
    Moves=[unknown_answer]
).
```

Question 1 is used as an example for filling in the important lines of the `answerQuestion` predicate:

```
answerQuestion(pq(zwemmen(pieter)),Models,Moves):-
    satisfy(zwemmen(pieter),Model,[],Result),
    Moves=[sensible_question,answerpq(Result)]
```

The `satisfy/4` predicate takes the declarative sentence which is the argument of the polar question operator and checks whether or not there is a way to satisfy it in the discourse-so-far. This will give a *neg* or *pos* as a result which will be stored in *Result*. *Moves* will then be assigned ‘sensible question’ and ‘`answerpq(Result)`’. These moves will then be used in the `realiseMove` predicate to generate the answers Curt gives the user.

```
realiseMove(sensible_question, 'Dit is een goede vraag').
realiseMove(answerpq(neg), 'Nee, dat is niet correct').
realiseMove(answerpq(pos), 'Ja, dat klopt').
```

4.2 Dutch Background Knowledge

As explained in section 2.2.4 there are three kinds of background knowledge in Curt. In Dutch Curt they have the following content:

- `lexicalKnowledge.pl`

The lexical knowledge of the portion of Dutch used in this thesis is built like the ontology visualized in figure 5. The top node is ‘thing’. The first disjunction is between organisms and objects. Objects are things like medals. Organisms can be divided in plants, animals and humans; because this domain only contains humans, the plants and animals are not mentioned. Swimmers are athletes and are humans just like man and woman are. But a man cannot be a woman and vice versa, therefore man and woman are declared disjunct. Proper names are found below the node declaring their sex. Lexical facts about verbs specify the kind of subjects/objects they take. The verb ‘krijgen’ has a person as a subject and a thing as a object.

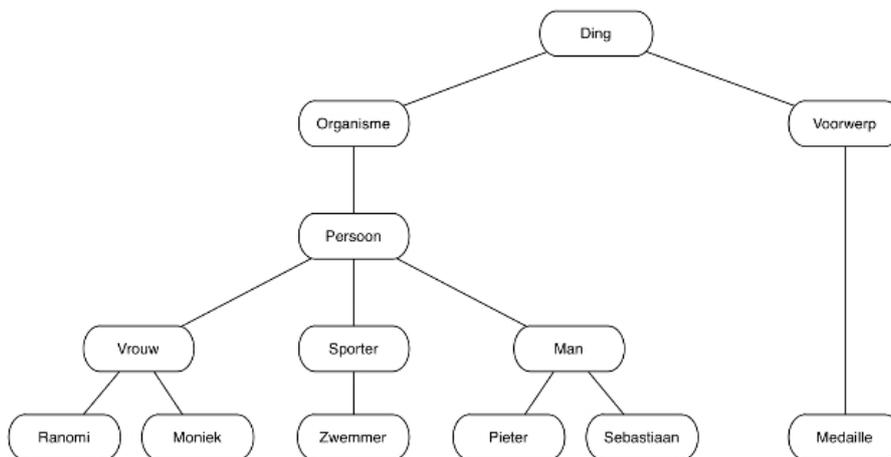


Figure 5: Ontology of the lexical knowledge of Dutch Curt.

A few examples from `lexicalKnowledge.pl`:

```
lexicalKnowledge(vrouw,1,Axiom):-  
    Axiom = all(A,imp(vrouw(A),persoon(A))).
```

```
lexicalKnowledge(moniek,0,Axiom):-  
    Axiom = all(A,imp(eq(A,moniek),vrouw(A))).
```

```
lexicalKnowledge(krijgen,2,Axiom):-  
    Axiom = all(X,all(Y,imp(krijgen(X,Y),and(persoon(X),ding(Y))))).
```

- `worldKnowledge.pl`

is the place for facts like ‘if a person is talking with another person, that other person is also talking with the first person’. In other words the symmetry of the verb `pratenmet`:

```
worldKnowledge(pratenmet,2,Axiom):-  
    Axiom = imp(pratenmet(X,Y), pratenmet(Y,X)).
```

- `situationalKnowledge.pl`

This file is empty because Dutch Curt has no restrictions of the form: ‘there need to be at least 2 people’.

4.3 Interacting with Dutch Curt

In this section the interaction with the result of this thesis is shown and discussed. All the elements discussed in sections 2, 3 and 4 come together in the system that is called Dutch Curt. It has the functionality of the original Helpful Curt created by Blackburn and Bos [2], in Dutch instead of English. As an extra feature, Dutch Curt can handle polar questions as well. To demonstrate Dutch Curt two short sample interactions are shown below.

Sample interaction 1. In this sample interaction shown in figure 6 Curt is told two things by the user. First ‘every woman is a swimmer’. The discourse-so-far is still empty so that poses no problems. Second ‘Moniek is talking with Pieter’. This sentence introduces `Moniek` and `Pieter` as two separate entities because in the `lexicalKnowledge.pl` file `Moniek` is declared female, `Pieter` is declared male and male and female are declared disjunct.

Then Curt is asked a question: ‘Does Moniek swim?’. The question is first parsed to obtain the semantic representation

```
pq(zwemt(moniek))
```

The body of the polar question operator is then entered into `satisfy/4`, the main predicate of `modelCheker2.pl`. To find out if there is a way to satisfy `zwemt(moniek)` in the discourse so far. This is possible even though it has not been spelled out by the user that `Moniek` is a woman nor that she swims. What happened is that `Moniek` is marked as a `woman` in the model due to the lexical information. That, combined with the introduced fact that ‘every woman swims’ leads Curt to the answer that `Moniek` is indeed swimming. Therefore Curt declares this a proper question and answers ‘yes’.

When the reserved command `models` is used the model Curt has built to represent the discourse so far is shown. `D=[d1,d2]` is the set of variables in the model. The rest of the model consists of various assignments for these variables. In the model, it can be seen that `Moniek` is indeed a woman, but also that she is `Moniek Nijhuis`, a `person`, an `organism` and a `thing`. In other words all the things that are higher up in the background knowledge ontology, see figure 5. Another thing that stands out is that not only is ‘Moniek talking with Pieter’ but also ‘Pieter is talking to Moniek’. This happens because in the world knowledge the verb `pratenmet` is declared symmetrical.

Sample interaction 2. In this interaction show in figure 7 the user enters two declarative sentences conveying that ‘Pieter congratulates Ranomi’ and ‘Sebastiaan congratulates Ranomi’.

Then the user tries to ask ‘Who is congratulating Ranomi?’. But the user mistyped and placed an extra ‘i’ at the end of Ranomi. Curt then asks the user what he/she meant, giving the user the opportunity to ask again.

The user proceeds to type the correct question, ‘Who is congratulating Ranomi?’. The question is first parsed to obtain the semantic representation, then the representation is changed to an existential formula in the `answerQuestion`:

```
que(A, persoon(A), feliciteren(A, ranomi))
some(A, and(persoon(A), feliciteren(A, ranomi))).
```

The existential representation of the question minus the existential quantifier is then used by the `satisfy/4` predicate to see if there is a way to satisfy the question. If that is the case a `findall` is employed to find all the ways to satisfy the existential representation of the questions. When that is done a correct answer is returned. Curt declares the question a proper one and answers ‘Pieter and Sebastiaan’ which is not only a true answer but also a complete answer to this question.

Then the `models` command is given and the model of the discourse-so-far is shown. Something strange has happened here, not only are `Ranomi`, `Pieter` and `Sebastiaan` in the model, `Moniek` is also in the model. This happens because `Ranomi` is declared in the `lexicalKnowledge.pl` file as a female, and not equal to `Moniek`. The only way to show this disjunction in the model is to declare them both separate entities, even though `Moniek` is not active in this discourse. Another thing to note is that `feliciteren` is not symmetrical in contrast with `pratenmet`.

```

?- curt.

> elke vrouw zwemt.

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK.

> moniek praat met pieter.

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK.

> zwemt moniek?

Curt: Dit is een goede vraag.

Curt: Ja, dat klopt

> models

1 D=[d1, d2]
f(0, moniek, d1)
f(0, moniekN, d1)
f(0, pieter, d2)
f(1, ding, [d1, d2])
f(1, man, [d2])
f(1, organisme, [d1, d2])
f(1, persoon, [d1, d2])
f(1, voorwerp, [])
f(1, vrouw, [d1])
f(1, zwemmen, [d1])
f(2, pratenmet, [(d1, d2), (d2, d1)])

> bye

Curt: Doeidoei!

```

Figure 6: Sample interaction showing declarative sentences and a polar question in Dutch Curt.

```

?- curt.

> pieter feliciteert ranomi.

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK..

> sebastiaan feliciteert ranomi.

Message (consistency checking): mace4 found a result.
Message (informativity checking): mace4 found a result.
Curt: OK.

> wie feliciteert ranomii?.

Curt: Wat zegt u?.

> wie feliciteert ranomi?.

Message (answer checking): prover9 found result "proof".
Message (answer checking): prover9 found result "proof".
Curt: Dit is een goede vraag.

Curt: pieter en sebastiaan.

> models.

1 D=[d1, d2, d3, d4].

f(0, moniek, d1).
f(0, moniekN, d1).
f(0, pieter, d2).
f(0, ranomi, d3).
f(0, sebastiaan, d4).
f(1, ding, [d1, d2, d3, d4]).
f(1, man, [d2, d4]).
f(1, organisme, [d1, d2, d3, d4]).
f(1, persoon, [d1, d2, d3, d4]).
f(1, voorwerp, []).
f(1, vrouw, [d1, d3]).
f(2, feliciteren, [(d2, d3), (d4, d3)]).

> bye.

Curt: Doeidoei!.

```

Figure 7: Sample interaction showing declarative sentences and a wh-question in Dutch Curt.

5 Conclusion and Future Work

This thesis had two main goals: first to convert the Curt system made by Blackburn and Bos [2] to Dutch, taking in account language differences; second to expand the Curt system with polar questions and the ability to answer those. To fulfill these goals Dutch Curt was created, a system that can do the following things:

- Parse Dutch declarative sentences.
Dutch Curt can parse several types of Dutch sentences. It can use proper names, nouns, definite and indefinite noun phrases, transitive and intransitive verbs, coordination, negation, relative clauses, and conditional sentences. The verbs are limited to third person singular.
- Evaluate these sentences for consistency and informativity.
When a sentence is parsed in Curt it is evaluated to see if it's fit to add to the discourse-so-far. First a consistency check is made, if the sentence is consistent with the discourse-so-far then a second check, one of informativity is made. If both checks are successful the sentence is added to the discourse-so-far.
- Parse and answer Dutch wh-questions.
Curt can parse who- and what-questions asked by the user, asking after the subject or object. The parsing renders the semantic representation of the wh-questions.
 - who-question: `que(A, persoon(A), intransitiveverb(A))`.
 - who-question: `que(A, persoon(A), transitiveverb(A,B))`.
 - who-question: `que(A, persoon(A), transitiveverb(B,A))`.
 - what-question: `que(A, object(A), intransitiveverb(A))`.
 - what-question: `que(A, object(A), transitiveverb(A,B))`.
 - what-question: `que(A, object(A), transitiveverb(B,A))`.

When a wh-question is parsed it has one of the first-order logic forms shown above. This semantic representation of the question is then used to answer the question. After verifying that there is an answer to the question asked, Curt employs a `findall` statement to find all the answers to the question asked. Curt then returns these answers to the user.

- Parse and answer Dutch polar questions.
Curt can parse polar questions asked by the user, that is questions that require a yes-or-no answer. The parsing renders the semantic representation of the polar questions. That is a the semantic representation of a declarative sentence adorned with the polar question operator:

`pq(Sentence)`

The *Sentence* is then evaluated in regard to the discourse-so-far using `satisfy/4`. If there is a way to make the *Sentence* true in the discourse-so-far then the answer to the question is yes, if not the answer is no.

This system is limited in several ways:

- The size of the lexicon and the background knowledge. The system is very small. New lexical items and lexical or other knowledge can be added relatively easily. However the system is not built for working with an enormous database so growth in that area is limited.
- The grammar deals only with verbs in the third person present tense. The grammar could be extended to include other persons, but there need to be found some solution for the agreement between subject and verb. Other tenses are hard and raise time issues that are a whole different study.
- More lexical types could be added for example prepositions and ditransitive verbs. Prepositions are important in everyday Dutch and would therefore be a good addition.
- Only who-, what- and polar questions can be answered by Dutch Curt. It would be feasible to extend Dutch Curt such as that it is able to handle other kinds of questions.

Because Curt is a highly modular system with some modules in Prolog and others in Perl. To extend the system and make it more user friendly one or more Prolog modules could be replaced with Perl modules. In Perl it's possible to create a graphical user interface (GUI) which would make Curt easier to use. The databases holding the lexical and grammatical information could also be stored in a Perl database. The parsing would then still be done in Prolog but looking all the pieces up in a huge database would then be processed by Perl, which is better suited for database handling.

References

- [1] D. Ahn, V. Jijkoun, K. Müller, M. de Rijke, S. Schlobach, and G. Mishne. Making stone soup: Evaluating a recall-oriented multi-stream question answering system for Dutch. *Multilingual Information Access for Text, Speech and Images*, pages 919–919, 2005.
- [2] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, 2005.
- [3] L. Hirschman and R. Gaizauskas. Natural language question answering: The view from here. *Natural Language Engineering*, 7(04):275–300, 2001.
- [4] W. McCune. Mace4 reference manual and guide. *Arxiv preprint cs/0310055*, 2003.
- [5] W. McCune. Release of Prover9. In *Mile High Conference on Quasigroups, Loops and Nonassociative Systems, Denver, Colorado*, 2005.
- [6] S. Stymne and Y. Samuelsson. A Swedish Curt Computational Semantics Term Project GSLT autumn.

6 Appendices

The appendix contains two parts. Section 6.1 explains where to find and how to install Dutch Curt and section 6.2 contains a short description of how to interact with Curt.

6.1 Installing Dutch Curt

To work with the Curt family you will need a Linux computer. The easiest way to ensure everything works as it should be, is to start with downloading the Curt family and the automated reasoning tools that are suitable to your needs. Then when that is up and running follow these steps:

- Download the zip-file containing Dutch Curt.
- Unpack the zip-file.
- Open a terminal and the directory in which you unpacked Dutch Curt.
- Start Prolog (usually with the `pl` command).
- Consult `dutchCurt.pl` and you are ready to go.

Blackburn and Bos:

<http://homepages.inf.ed.ac.uk/jbos/comsem/software1.html>

A Dutch Curt:

http://dl.dropbox.com/u/40475750/Logic-based%20Question%20Answering%20in%20Dutch/A_Dutch_Curt.zip

6.2 Working with Curt

To start Dutch Curt simply type in ‘curt.’ After Curt is started you can type in a sentence or question. These sentences and questions are possible with and without capital letters and with and without full stops and question marks. Typically you would first type in some sentences to build up a discourse and then question the discourse. To show all the options you could type ‘help’. This will give you the following list with options:

- readings: prints current readings
- select N: select a reading (N should be a number)
- new: starts a new discourse
- history: shows history of discourse
- models: prints current models
- summary: eliminate equivalent readings
- knowledge: calculate and show background knowledge
- infix: display formulas in infix notation
- prefix: display formulas in prefix notation
- bye: no more talking

All the commands are described above are reserved commands. They are not parsed as the rest of the sentences. Try some of the examples from this thesis or have a look at `dutchLexicon.pl` to see what other sentences/questions you could make.