Willy

--------------------------
the intelligent agent

Project 1
Artificial Intelligence, VT04
Göteborgs universitet

Group ai04-14:

Håkan Burden
c1hburd@cling.gu.se

Carine Cassia
c1ccass@cling.gu.se

Emma Olsson
c1eolss@cling.gu.se

Anna Wählby
c1awahl@cling.gu.se
Abstract
We have implemented an intelligent agent called Willy from the theories of BDI, Beliefs, Desires and Intentions. Our agent is goal based: to give the user a recipe based upon the users wishes about ingredients, preferences and number of portions. The agent is aided by a key parser and a database of recipes. The program and the database are written in Prolog.

File direction
Our files is in our course account ai04-14 under
~/project/program/’files’

How to run our program
You simply run our program by printing

sicstus -l main.pl --goal willy.

in your x terminal. Or by consulting main.pl in Emacs.
Content

1 Introduction .............................................................................................................. 1

2 Thoughts behind our program .................................................................................. 1

3 Our program ............................................................................................................. 1
   3.1 File description .................................................................................................. 1
   3.2 The intelligent agent ......................................................................................... 3
       Automatic reasoning - the art of perceiving ......................................................... 4
       Acting rationally ................................................................................................. 5
   3.3 Natural Language Processing .......................................................................... 6
       The Lexicon ....................................................................................................... 6
       The Parser ....................................................................................................... 7
   3.5 The database .................................................................................................... 8
   3.6 Searching the database .................................................................................... 9
   3.7 How the searching works .................................................................................. 10
   3.8 Handling restrictions ....................................................................................... 11
   3.9 Feedback and Mutual Beliefs ........................................................................... 12
   3.10 Calculate portions ......................................................................................... 12

4 Discussion ................................................................................................................. 13
   4.1 Restrictions in our system ............................................................................... 13
       The BDI model ................................................................................................ 13
       Parser and lexicon ......................................................................................... 13
       Search for ingredients ...................................................................................... 13
       Hyponyms ...................................................................................................... 13
       Calculating portions ....................................................................................... 14
   4.2 Future development ......................................................................................... 14

Appendix: Example dialog from the program
1 Introduction

Many of us have been standing in the supermarket, hungry and tired after a day of work and couldn’t decide what to eat. Maybe you have some ingredient you want or something is really cheap today. What you need is the intelligent agent Willy who gives you a recipe based on your wishes. He is a text based intelligent agent who delivers recipe suggestions.

2 Thoughts behind our program

Our approach in this project was not to design a parser in an elaborated NLP-interface but to construct an intelligent agent.

The agent is intelligent from the perspective of acting rationally. It does not think in any way we would recognize as human thinking but it has representations of mental states. From a certain mental state the agent is triggered to perform a corresponding action. Acting rationally is the skill of performing the action that will make the agent reach its goal; or if that is not possible, to reach a subgoal as close to the goal as possible.

It also means that the agent has some kind of means for representing the discourse between agent and user. Except knowledge of the user the agent also has resources for meeting the will of the agent. This knowledge is then used in order to decide which action will be the most rational.

In our domain the rationality is shown in the intention of the agent to return a recipe fulfilling the wants of the user. It is a goal based agent. To be able to do this the agent "keeps in mind" what the user has specified for ingredients, number of portions and if there are any other restrictions, like vegetarian or no lactose. This means that our agent is always striving for exchange of information. Anything else is beyond its capabilities.

In order to implement these features of the agent we have used the BDI-theory, were BDI stands for;

- Beliefs of the agent.
- Desires to know more
- Intentions to in accordance to Beliefs and Desires act in a rational way

In addition we also use the concept of Mutual Beliefs what the agent knows is shared knowledge with the user.

Everything will be implemented in Prolog.

3 Our program

3.1 File description

Out program consists of:

The Main program: The main.pl is the file that connects the NLP, Natural Language Processing, and the agent.

The dialogue loop is managed by the talk/1 predicate, the variable is either on or off. Thus is the dialogue handled to react to the will of the user. It is based on the idea of intertwining dialogue. Every
input from the user is processed so that an output can be returned. This in turns spurs a new input from the user.

It uses the files:

- swedishReadLine.pl for user prompt and tokenisation.
- keyparser.pl to extract key words from user input.
- agent.pl to perceive and act on the input.
- random_output.pl to produce randomised greetings.

**The Agent:** This part of the program is the intelligent agent based on the BDI-theory. The file agent.pl has two key predicates perceive/2 and action/3. It uses the files:

- full_search.pl for giving a final recipe
- part_search.pl for consistency checking
- name_search.pl searching using a recipe name
- feedback.pl for giving feedback and producing new questions, feedback is using the file random_output.pl to produce random greetings.
- portions.pl for calculating new portions in the recipe
- conv_restrictions.pl for converting restrictions into ingredients that shouldn’t be included. conv_restrictions is using the file lexicon.pl to convert restrictions.

**The NLP:** The NLP consists of the keyparser.pl and the lexicon.pl. keyparser.pl takes an input of words and makes a semantic interpretation of the user input. lexicon.pl is our lexicon.

**The Database:** recipes.pl is the recipe database; it is searched by the methods in the files full_search.pl, part_search.pl and name_search.pl.
3.2 The intelligent agent

Our agent is called Willy. The name is taken from a recipe program some of us made in our course of logic programming. It is from the same program the database is taken.

As we already mentioned our agent is based on the BDI approach. In our specific case Belief has to do with what the agent ‘knows’ about the different information it got from the user; that is, what ingredients are to be in the recipe or how many people to serve. The Desire part is what the agent wants to get information about. They are asserted when we start the program:

\[
\begin{align*}
\text{willy}(\text{des/ask(restr)}). \\
\text{willy}(\text{des/ask(ingred)}). \\
\text{willy}(\text{des/ask(port)}).
\end{align*}
\]

This translates to the desire to ask for restrictions, ingredients and portions. When the dialog is running they will help the agent to choose what action to use.

In this implementation we didn’t use the Intention part explicitly since they are actually implicitly represented when the agent goes from a set of desires and beliefs to performing the corresponding action.

The key predicate of the agent is agent/3.

\[
\text{agent}(\text{Sem}, \text{Out}, \text{on}) :-
\begin{align*}
\text{perceive}(\text{Sem}, \text{Name}), \\
\text{action}(\text{Name}, \text{Conds}, \text{Effects}), \\
\text{do_list}(\text{Conds}), \\
\text{do_list}(\text{Effects}), \\
\text{formulate}(\text{Name}, \text{Out}).
\end{align*}
\]

The first argument is the semantic interpretation from the parser, the second the output to the user and the third decides if the dialogue is on or off. There are some special cases when the Sem variable contains specific key words that are treated without going through the perceive and action part. Being able to cut straight to formulate saves time. For instance it is not necessary to in any way save the fact that the user wishes to quit, just to formulate a suitable output and let talk/1 in main.pl know the dialogue is off.

Our agent uses both system and user initiative. The default setting of the agent is not to straightforwardly connect the perceiving (automatic reasoning) of the input to the performing of actions and output. If everything works out correctly the Name variable will not be instantiated before the action/3 call.

It is the set of beliefs and desires in a mental state that determine which action that is going to be the next move. In some cases the input will determine next action but it could just as well be that the information in the input is stored for later use and the agent asks for more urgent information.

For instance it could be that the user answers the question of an ingredient with the number of portions. This is no problem; the agent stores the information and asks again for ingredients. On the other hand it could be that the question is for portions and then the answer fits well with the prerequisites for returning a recipe to the user.

A layout of this kind makes the dialogue run smoothly. Answers can be informational overloaded without causing problems, extra information is stored and the questions supposed to trigger the corresponding answer from the user will just not be asked. Underloaded answers will in turn trigger new questions for the missing information.
Due to the BDI structure our agent can use any kind of system-initiative. If the user is unsure or not used to the system the agents dialogue structure will lead to closure. Otherwise the user can take initiative and give more information than required and end up with the recipe much faster. It is also possible to interrupt the ongoing dialogue to quit or restart. This enables the agent to toggle between user initiative, system initiative and mixed initiative.

Automatic reasoning - the art of perceiving

All information from the user is passed through the parser and converted to the internal representation language. The agent in turn perceives this. While the information is being perceived the agent checks that its beliefs are not inconsistent. For instance it cannot believe both \texttt{ingred(salt)} and \texttt{no(ingred(salt))}. It also runs a consistency check with regards to the ingredients every time it has perceived the input from the user. The agent will not go on asking for more information if there is no recipe based on the available information. It will instead ask the user to change some of the information.

The new information gets treated in different ways depending on its context. An answer is interpreted differently depending on the question that ignited it. Compare:

\begin{verbatim}
Q: Are there any restrictions? |  \square X \downarrow \square \texttt{ingred(X)} \\
A: Salt | \hspace{1cm} X = \texttt{salt} \\
I: No salt in recipe | \hspace{1cm} \texttt{no(ingred(salt))}
\end{verbatim}

\textbf{to}

\begin{verbatim}
Q: Do you want something else in your recipe? |  \square X \downarrow \texttt{ingred(X)} \\
A: Salt | \hspace{1cm} X = \texttt{salt} \\
I: Salt in recipe | \hspace{1cm} \texttt{ingred(salt)}
\end{verbatim}

So in order for the agent to make the right conclusions of received information it is vital to know the context. Another example is if a single number is perceived and the last move of Willy was to ask for number of portions. Then the number will be interpreted as the number of portions. Otherwise it will be ignored, since we do not account for number of bananas or onions to be in the recipe.

\begin{verbatim}
perceive([count(Key)|Keys],_):-
  willy(last_move(ask_port)),
  asserta(willy(bel/port(Key)),
  perceive(Keys,_).
\end{verbatim}

Other possible perceptions:

- assert a recipe.
- identify number of portions, there is no longer a desire to ask for portions.
- change \texttt{no(ingred(I))} to \texttt{ingred(I)} and vice versa.
- assert an ingredient or its negation.
- remove contradicting information concerning ingredients.
- convert a restriction like vegan to a number of implicit restrictions: \texttt{ir(ingred(I))}.
- retract a restriction and its implicit restricted ingredients.

After running a dialogue with Willy the agent will have a certain mental state. For example:

\begin{verbatim}
willy(bel/ingred(lök)).
willy(des/ask(port)).
willy(last_move(ask_ingred2)).
\end{verbatim}
As long as the information is consistent the perceive predicate will not interfere with what the next action will be. But if the information given by the user is inconsistent or the agent has got something wrong it will notify the user. This happens when a set of ingredients to be or not to be in the recipe are unrealisable. The agent then has no choice of actions but has to question the user until a consistent set of ingredients is at hand.

\[
\text{perceive([], quest_ingred)} :- \\
\text{\+ part_search,} \\
\text{asserta(willy(des/quest_ingred)).}
\]

Usually the second argument of the perceive predicate is an unknown variable.

Another situation when the agent is forced to act in a certain way is when the user has answered 'Yes' to the questions 'Anything else?' and 'Anything to be excluded?'. The user is asked to specify its response.

\[
\text{perceive([yes], specify):-} \\
\text{(willy(last_move(ask_ingred2))} \\
\text{; willy(last_move(ask_restr))} \\
\text{), asserta(willy(des/specify)).}
\]

**Acting rationally**

In order to return a recipe from the information given by the user the agent tries to perform the action named give_rec:

\[
\text{action(give_rec,} \\
\text{[willy(bel/ingred(_)),} \\
\text{willy(bel/port(_)),} \\
\text{\+willy(des/ask(restr))]}, \\
\text{[full_search(Rec),} \\
\text{right_port(Rec,Nrec)}, \\
\text{asserta(willy(bel/found(Nrec))),} \\
\text{retract(willy(last_move(_))]), \\
\text{asserta(willy(last_move(give_rec))))].}
\]

Every action has a name, a list with conditions and a list with effects to perform. In order to succeed do_list/1 has to check that all the facts in the list of conditions are true. If so it can continue by performing the calls in the list of effects.

If we take for example the same action give_rec we can describe it as following:

- **Name:** give_rec
- **Conditions:**
  - it has to have at least one ingredient.
  - it has to know for how many people the recipe is intended.
  - it should not have the desire to ask about restrictions.
- **Effects:**
  - it will find a recipe that fits the information using full_search/1.
  - it will calculate the right amount for the ingredients in the recipe.
  - it will change the argument of last_move to give_rec.

Other possible actions:

- finding a recipe by a name given by the user (if there are no mentioning of portions in Willys memory the recipe is returned with original amounts to ingredients)
- ask the user for more specific information on more ingredients and restrictions
- question the ingredients given by the user if they are inconsistent, and make sure that the desire to ask for more ingredients is at hand
• ask for a first ingredient
• ask for more ingredients
• ask for a restriction
• ask for number of portions
• inform user that it is not possible to find more recipes from given information

The order of the actions is important since Prolog will try with the first listed and if it fails with the preconditions it will backtrack to try next action. Therefore we have tried to have the actions returning recipes as high as possible and the ones asking for more information last. All in the spirit of the agent: as quickly as possible find a recipe.

The last thing Willy does is to formulate an output were any possible feedback will given with the intention of the new action.

3.3 Natural Language Processing

To handle the input and the output of the dialogue we reused the file swedish readline.pl from the course formell semantik we had last autumn. This file is inspired from Blackburn and Boss s’ readline.pl and has been modified to be able to handle the Swedish letters ’å’, ‘ä’ and ‘ö’. The predicate readline/1 provides a prompter and reads a sequence of character from the input, and converts this to a list of atoms. The punctuation is stripped. This representation of the input then goes to the parser.

The Lexicon

To be able to understand the input words we need a lexicon that states information about the words. In our lexicon we have all the ingredients appearing in the recipes in the database, the full names for the recipes and all the words appearing in the recipe names or the descriptive names. In addition we have words for describing the restrictions like “lactosintolerant” and “vegetarian”. Some other useful words that are important for the dialogue have also been included, e.g. “quit”, “yes” and “no”. But since the focus hasn’t been on the parser in this project, the lexicon didn’t need to be sophisticated enough to cover more verbs and functional words.

We have followed the same form as a lexicon formerly made by Peter Ljunglöf, Chalmers University, that we used during the course parsing algorithms. But we have also added a part that gives semantic information about the word, an idea we got from Blackburn and Bos’ material for semantics and Prolog.

There are roughly 500 words in the lexicon.

The words are classified in different predicates stating their part-of-speech: adjektiv0/3, adverb0/3, artikel0/3, interjektion0/2, räkneord0/3, substantiv0/3, verb0/3. This translates to adjective, adverb, article, interjection, countable, noun and verb.

There are three different structures for the word-predicates in the lexicon, but they are built up in the same:

• All the predicates contain the word represented as a string.
  räkneord0("ett").

• The last argument for both the 2- and 3-termed predicates is a set of words that gives some semantic information describing the word or expression.
interjektion0("hej då", [quit]).

- For the 3-term predicates the second argument is a term, f/4, that gives grammatical information about the form of the word. This information is genus, numerus, definite form and casus. But this has not been used in our implementation.

  substantiv0("champinjoner", f(utr,plu,ind,nom), [ingred, vegetable, mushroom]).

The word can be a string with one or more words, since they are compound names. For example we represent full recipe names that contain more than one word as a noun. E.g.

  substantiv0("karlalfreds dundersoppa", f(utr,sin,ind,nom), [recipe]).
  substantiv0("gul lök", f(utr,sin,ind,nom), [ingred, vegetable]).

For most words we have given a list (more correctly a set) of semantic information. Some articles and interjections are for example marked as [neg] for negation, some adjectives and nouns as [restr] for restriction.

The two main groups are ingred and recipe telling that the word is either an ingredient or a recipe. For the ingredients there are also a few hypernyms that we have used to mark up the word. In this way we are able to group ingredients that belong together. dairy, ovo, vegetable, mushroom and nuts are some of the hypernyms that we use.

These can then be used to extract information about the recipes based on the facts of their ingredients. This is the reason we choose to include the semantic information about the ingredients in the lexicon instead of for example straight in the recipe-database. To have this information in the lexicon is reflecting the way that our human mind works since we ourselves would know that yoghurt includes milk and this information would normally not be stated in a recipe book.

The Parser

The list of atoms from swedishReadLine.pl is first transformed into a list of strings with the predicate token/2, in order to suit the representation of the lexicon.

Our parser is more of a keyword extractor that goes through the input words and extracts all the words the system understands -that is the ones that are in the lexicon. Instead of parsing multiword phrases as tree structures we check the input for short sequences of words that appear in the lexicon (since we have stored multiword ingredients and recipe names). In other words it gives a simplified representation of what the user said. It can handle different kinds of information:

<table>
<thead>
<tr>
<th>Kind of information</th>
<th>Some examples of translations</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ingredients which have to be in the recipe</td>
<td>ingred(&quot;potatis&quot;)</td>
</tr>
<tr>
<td>A recipe name</td>
<td>rec(&quot;frasses våfflor&quot;)</td>
</tr>
<tr>
<td>Restrictions</td>
<td>restr(&quot;vegan&quot;)</td>
</tr>
<tr>
<td>Negation</td>
<td>no(ingred(&quot;mjölk&quot;))</td>
</tr>
<tr>
<td>The needed portions</td>
<td>port(3)</td>
</tr>
<tr>
<td>Diverse command words</td>
<td>quit, restart</td>
</tr>
</tbody>
</table>

To find out which kind of information it got, it uses the hypernym list that is in the lexicon as in the following example:

```prolog
keyParse([Input|InputRest], [ingred(Input)|OutputRest]):-
  substantiv0(Input, _, [ingred|_]),
```

7 (14)
keyParse(InputRest, OutputRest).

When it has found the word in matter, Input, it looks at the hypernym list that follows and if the first one is ingred then it will be represented as ingred(Input). This can be applied to information such as recipe or restriction as well.

It can also handle sequences of words such as “gul lök”. It has to append these two words first to then be able to match them with the lexicon. It looks like this:

keyParse([Word1, Word2|Words], [ingred(String)|OutputRest]) :-
  append(Word1, " ", Str),
  append(Str, Word2, String),
  substantiv0(String, _, [ingred|_]),
  keyParse(Words, OutputRest).

As the portions are concerned the parser can handle both digits like 2 and spelled out numbers “two”. If the input has a number followed by a word such as person or portions it will be interpreted as port(Digit). If no such word is found it will only be interpreted as count(Number). The agent will take care of it later.

To handle negation we just described the different combinations of words that we thought were possible to have as input from the user.

- inte/inga potatis
- inte/inga gula lökar
- inte ha potatis
- inte ha gula lökar

Finally the parser can handle different command words that receive different semantic representation:

- nej becomes [no], ja [yes]
- words like ‘hejdå’, or ‘sluta’ become [quit]
- ‘börja om’ becomes [restart]
- ‘nytt’ becomes [new]

The parser ignores all other input that is not in the lexicon. It means that you can write things like “grrrr potatis flop” and it will be able to interpret ingred(“potatis”). But one of the advantages is that it gives much freedom to the user in her/his wording.

It has no problem to manage much information at the same time. Look at the following example:

Willy: what do you want in the dish?
> We are 5 persons and I want to do something with potatoes.

The parser will interpret this as following:

[ port(5), ingred("potatoes") ]

3.5 The database

The database has 105 recipes. All the recipes are in Swedish. The database is taken from a recipe program called "Willys Matlåda" (www.cling.gu.se/~cl1eolss/matladan) from an earlier course. Because the database comes from a web based program where the user adds the recipes via the web it contains some spelling errors which we have not taken care of.
The database consists of different types of recipes: drinks, desserts, salads, meat dishes, cakes, fish, chicken, vegetarian dishes.

Every recipe contains a number of portions, a category (e.g. pasta, vegetarian, meat dish), a recipe name and a describing recipe name, a list of ingredients with amount, unit of measurement and ingredient name and a description of how to make the recipe. The recipes are of the format:

```
recept(
    portioner(NumberOfPortions),
    category,namn(RecipeName, AlternativeName),
    ingredienser([ingred([Amount],Ingredient)]),
    beskrivning([DescriptionLines])
).
```

We have modified the original database to make it fit our needs. The ingredients amount was initially a list `ingred(["1 dl vatten"])` but we have separated the amount, the unit of measurement and the ingredient into `ingred([1, dl], "vatten")`, to make it easier to recalculate the amount and search for an ingredient.

Example recipe from our database:

```
recept(
    portioner(2),
    soppor,
    namn("karlalfreds dundersoppa", "spenatsoppa"),
    ingredienser([
        ingred([500, gram], "spenat"),
        ingred([1, tärning], "grönsaksbuljong"),
        ingred([1], "vatten"),
        ingred([], "salt"),
        ingred([], "peppar"),
        ingred([1, klick], "smör"),
        ingred([2], "ägg")],
    beskrivning(["Smält den frusna hackade spenaten i en gryta, tillsätt en buljongtärning.",
    "Späd med vatten till önskad konsistens.",
    "Krydda och låt soppan koka fem minuter.",
    "Precis innan soppan är klar tillsätts en klick smör.",
    "Servera med kokta ägg i klyftor."])).
```

3.6 Searching the database

The agent has different search needs for different situations. There are three situations when the agent wants to search the database:

1. The user has given a recipe name as input (Name search)
2. When the agent has received information from the user, the agent wants to know if he should proceed with the information he has or if something needs to be altered. This consistency check is used after every new input from the user (Partial search).
3. The agent has the desire to give a final recipe (Full search).

We have made three different search predicates, to fit the search situations and to make the search as efficient as possible.

The search methods uses the agents beliefs `willy(bel/X)` when searching through the database.
3.7 How the searching works

- **Full search** (Predicate `full_search` in the file `full_search.pl`): Searches through the whole database using the information about the ingredients `willy(bel/ingred(X))`, ingredients not to be included `willy(bel/no(ingred(X)))` and implicit restrictions `willy(bel/ir(ingred(X)))` the agent has asserted and returns a random recipe that fits the agent's beliefs.

  The full_search works as a filter. It takes the first ingredient and puts all the fitting recipes in a recipe list. The recipe list is then filtered first by every asserted ingredient and the result is then filtered by every restricted ingredient. The ingredients `(willy(bel/ingreds(X)))` and the restrictions `(willy(bel/no(ingred(X))) and willy(bel/ir(ingred(X))))` is fetched from the Prolog memory and added to a ingredient list and a restriction list. The filtering shown below is done for both the lists, except that the recipes are thrown away if they have the ingredient in their ingredient list when filtering through the restriction list.

  The recipes in the database
  
  rec 1  
  rec 2  
  rec 3  
  rec 4  
  .  
  .  
  rec n  
  are made into a list

  All the recipes from the database that have the first ingredient in their own ingredient list

  | rec 1 | rec 6 | rec 7 | rec 18 | rec 34 | rec 69 | rec 78 | rec 87 | rec 101 |

  All the recipes from the recipe list that have the second ingredient in their own ingredient list

  | rec 1 | rec 6 | rec 69 | rec 78 | rec 87 | rec 101 |

  [recursive call for all the ingredients in the list]

  All the recipes from the recipe list that have the last ingredient in their own ingredient list.

  | rec 1 | rec 78 | rec 101 |

  A random recipe is returned from the final recipe list

  **Illustrating example of the filtering**

- **Partial search** (Predicate `part_search` in the file `part_search.pl`): Searches through the database after the first recipe that fits the agent’s beliefs, if it finds a recipe the predicate is successful.

  The partial search takes a recipe from the database and checks if all the current beliefs about ingredients `(willy(bel/ingred(X)))` are in the recipe. And it also checks that none of the not wanted ones `(willy(bel/no(ingred(X))) and willy(bel/ir(ingred(X))))` are not present. If one of the ingredients is not in the recipe or if one of the restrictions are in the
recipe, a new recipe is picked from the database. This is continued until a fitting recipe is found.

- **Name search** (Predicate name_search in the file name_search.pl): Takes a recipe name and returns the recipe. It is used when the user asks for a certain recipe (the parser has identified the input as a recipe name). The method checks if the input recipe name is a substring of either the recipe name (RecipeName) or the describing recipe name (AlternativeName) in the database namn(Name, AltName).

The partial search method has a worse case that is very inefficient, if it has to go through all the recipes for all the ingredients to find a fitting one or to fail. The full search method will always go through all the recipes for one ingredient and the number of recipes will in worst case scenario never shrink. Then the recipes will be looked at as many times as ingredients to consider while searching.

We decided to do two different methods because in the most cases when the agent does a consistency check it will find a recipe easily, so going through the whole database would then hopefully be unnecessary.

### 3.8 Handling restrictions

If you say to the Agent that you are "laktosintolerant", "vegan" or "vegetarian" it is asserted as a restriction willy(bel/restr(Restr)) in the keyparser. We are then using the hypernym list in the lexicon to assert all the ingredients that has the corresponding hypernym in its list. For instance lactose intolerant uses the hypernym dairy. The restrictions are matched with the hypernyms we use in the lexicon. For example if the user has said "vegetarian" we translate it to the hypernyms meat and fish that is used in the database, which means that all ingredients with meat and fish in their hypernym list should not be included in the recipe.

```prolog
restr_to_NoIng("vegetarian") :-
    doall(hypo_to_NoIng(meat)),
    doall(hypo_to_NoIng(fish)).
```

The hypernyms (ingredients in the lexicon) for every hypernym are asserted as willy(bel/ir(ingred(Hyponym))) so the agent will know which ingredients shouldn't be in the recipe. These implicit restrictions (ir) are then treated the same as willy(bel/no(ingred(X))) in the search methods.

```prolog
hypo_to_NoIng(Hypo) :-
    substantiv0(Ing, _, HypoList),
    member(Hypo, HypoList),
    "+ willy(bel/no(ingred(Ing))),
    asserta(willy(bel/ir(ingred(Ing)))).
```

We separate implicit restrictions from no ingredients because of the feedback. We don’t want to give feedback on all the ingredients the restrictions are translated into, only the ingredients the user explicit says and the restriction itself. We solved this problem by naming them differently and treating implicit ingredients and no ingredients the same in the search methods.
3.9 Feedback and Mutual Beliefs

To give the user a chance to correct mistakes right away and to know what the agent has perceived we use feedback. If the user is not pleased with the feedback he/she can correct the information he/she has given.

We use embedded implicit feedback, so that the feedback is embedded in the next utterance. This type of feedback doesn’t force the user to respond to the feedback. But the user can respond if wanted. If the agent doesn’t get any response on the feedback it will go on as usual.

We use beliefs and mutual beliefs to generate the feedback. The agent gives his new beliefs as feedback. Beliefs once given as feedback are then mutual belief, willy(mbel/X). Knowledge shared by the agent and the user. Information that is mutual belief won’t be given as feedback again.

There is one exception, when feedback is given with the ask_restr question. Then all ingredients (even those considered mutual beliefs) will be given as feedback.

Example dialog (translated):

Willy: what do you want in the dish?
> We are 5 persons and I want salt.

The agent has now asserted willy(bel/port(5)) and willy(bel/ingred("salt")).

willy: You are 5 persons. You want salt. Is there anything else you want in the dish?

The agent has now asserted willy(mbel/port(5)) and willy(mbel/ingred("salt")) so it will not be given as feedback again.

> I am a vegetarian and we are now 4 persons.

The agent has now retracted willy(mbel/port(5)) and asserted willy(bel/port(4)) and willy(bel/restr("vegetarian")).

willy: You are 4 persons. I consider vegetarian. Is there anything else you want to exclude?

The agent has now asserted willy(mbel/port(4)) and willy(mbel/restr("vegetarian")).

We also use mutual beliefs when the agent has given a recipe. When a recipe has been found it is asserted as willy(bel/found(Rec)). This is then printed out and when it has been printed it is asserted as willy(mbel/rec(Rec)) so that the agent wont suggest the same recipe twice if the user wants another recipe.

3.10 Calculate portions

Every recipe has an argument port(Port). If the portions doesn’t match the desired portions the agent calculates all the amounts ingred([Amount, Measurement, Ingredient]) in the recipe to fit the desired amount. If for example the desired portions are 4, but the found recipes portions are 2, the agent goes through all the ingredients in the ingredient list and multiplies it with 2. This is done for all ingredients where the amount list starts with a number, otherwise the list is left unchanged (e.g. ingred([lite], “salt”)).
In some cases the new amount will have 16 decimals. this is a bit much so we simply cut the decimals down to 1 without rounding off.

4 Discussion

4.1 Restrictions in our system

The BDI model
To fully use the BDI-model the search methods should search using the agent’s mutual beliefs, what the agent and the user has agreed on, but we are searching using the beliefs. We are only using mutual beliefs for feedback. After a belief has been given as feedback it’s becomes a mutual belief. But not all beliefs are given as feedback. In the end of the dialog (see appendix) feedback isn’t given, for instance after when asking for portions. If not everything is given as feedback, not everything becomes a mutual belief and it would then be missed when searching. Looking back it would have been better and more correct to give feedback on everything and then search on mutual belief instead of beliefs.

Parser and lexicon
As we have said earlier we have a very limited parser. The parser searches for one or two word combinations to be found in the lexicon (e.g. multiword ingredients or recipe names). We could have developed a more systematic method for both two-words and longer word combinations search. But since this was not prioritised the parser can’t find longer recipe names like “Annas fishy säs”.

Since the parser is based on the lexicon it’s actually the lexicon that sometimes is limiting the parser. At first we included in the lexicon only the ingredients that you can find in the recipe database. But if you said that you wanted to cook something with an ingredient that was not in the lexicon Willy was only able to say ”what?”. To solve this problem we thought that we could add some ingredients that we knew were not in the database, such as ”strawberry”. At first we thought this was a good solution because Willy was able to say “we don’t have any recipe with strawberry in”. We noticed later that if you say that you want to cook something with both strawberry and raspberry you get Willy’s answer: “I don’t have any recipe with both raspberry and strawberry, which of these two ingredients do you want to change?”. You never get the information that there actually are no recipes with strawberries and that you must change to an existing ingredient.

Search for ingredients
When we search for ingredients we use exact string matching. Which means that if I have asked for a recipe with onions the system won’t find the recipe with yellow onions. This is a quite greedy method that might exclude some recipes that we could be interested in. This could be solved using a search for substrings instead, (as we do with the recipe names), that would give a more generous result. The reason why we’re not doing that is that other problems will appear. The generous method would include recipes that we don’t want, (e.g. give recipes with eggplant at the search for egg, or milk chocolate when we search for milk), and we find this a greater problem than the first.

Hypernyms
In our program we used the hypernyms from the lexicon to understand a vegetarian as not wanting any ingredients classified as meat or fish. This was easy as we could treat them as negated ingredients in the search algorithm. It is also possible to state a group of ingredients like nuts and say that you are allergic to them. The interpretation is that nut is a restriction and this works fine; you won’t get any
recipes with any kind of nuts (as long as the information in the lexicon is correct). One problem with this is that we have only used the hypernym nuts to represent a restriction from a group of ingredients, so even if the user says that he wants nuts in his food, this is wrongly interpreted as a restriction, and no nuts will be included. (See discussion on Future development 4.2)

Another problem appears when you state something as a specific ingredient that is not a hypernym to other ingredients. “I’m allergic to tomatoes” will be interpreted as no(ingred(“tomatoes”)), and since the search checks for the exact string “tomatoes” all the other tomato-words like “sun dried tomatoes” and “tomato puree” won’t be excluded, which could be very annoying for the user who would probably lose some faith in Willy’s capability and intelligence. The solution could be to add more hypernyms to the lexicon that would state all tomato-ingredients as tomato. And also make sure that Willy really knows what he’s doing, so that he won’t think that he knows what an allergy is when he doesn’t understand the full implication of what it means to be allergic to something, like tomatoes.

Calculating portions
Prolog has many benefits but arithmetic is not one of them. Are solution to decimals was easily implemented but is crude and does not make the numerical values that very good looking.

4.2 Future development

We have used the hypernyms in the lexicon to make restrictions of a whole group of ingredients, but it could also be used for the opposite; to say that you do want any of the ingredients in a group. I really wish that I could say that I want chocolate, and this would be interpreted as all the ingredients having cocoa as a hypernym, and that I would like to have any of those ingredients in my recipe. If we extended the hypernym list for the ingredients this could also solve the searching problem mentioned above. For example onion could be used for both yellow onion, red onion spring onion etc.

Another way of solving the problem is to transform input words to their lemma and use the lemmas for searching in the lexicon and the recipes.

A better parser would be good to have with a lexicon covering a bigger portion of Swedish. A parser that at least understands noun phrases, and outputs the same kind of semantic representation to the agent. The agent wouldn’t have to change even though another kind of parser would be added. Implementing a full parse would demand a way of treating syntactical and lexical errors.

The semantics received from a parsing with a proper parser would have the same hierarchical tree structure as the syntax. This is no problem: it only means that we would need to extract the semantics that Willy demands from the tree structure.

Another extension for Willy would be to add a speech recogniser. As with a better parser the agent could remain in the same way as long as the speech recogniser had the correct semantic output. The dialog between the user and Willy could well be in spoken form, Willy using a nice text-to-speech application, but the final output from Willy; the recipe, is probably better shown in written form. This program could then be used on a mobile phone/ palm pilot, easy to access when wanting inspiration in the supermarket!
Appendix – example dialog from the program

---

Willy

by Burden, Cassia, Olsson and Wählby

MA students of Computational Linguistics

Gothenburg University

?- willy. - If you want a dialogue with Willy

---

% consulted /users/cling/cling01/clleolss/vt04/ai/projekt/willy/main.pl in module user, 120 msec 21584 bytes
% source info
| ?- willy.

Willy: Hej. Hur kan jag hjälpa dig?

> Hej jag vill laga något med potatis.

Willy: Ska det vara nåt mer än än potatis?

> ja gärna lök

Willy: Du vill ha med lök och potatis. Är det nåt som inte ska vara med?

> jag är laktosintolerant

Willy: Hur många portioner?

> 6


Detta recept har jag hittat:

grönsakspakoras (grönsaksbiffar)

Kategori: vegetariskt

6 portioner

Ingredienser:
1.5 stor potatis
1.5 litet blomkålshuvud
1.5 liten röd paprika
1.5 medelstor lök
3.0 blad vitkål
2.2 dl majs
4.5 tsk garam masala
3.0 tsk koriander
6.0 dl mjölk
1.5 tsk bakpulver
1.5 kniveudd chilipeppar
1.5 msk citron
matolja

Beskrivning:
Skala och köka potatisen nästan mjuk, koka kål och blomkål i ca 7 minuter. Hacka alla grönsakerna mycket fint.
Rör ihop kryddor, mjöl, presad citron och bakpulver tillsammans med kallt vatten tills det blir en ganska tjock smet.
Blanda i allt grönsakshack i smeten. Värma en stektpanna med olja och fritera matskedstora plättar i oljan utan att de blir gyllenbruna, vänd dem i oljan och servera dem till mango chutney.

Är du nöjd med detta recept och vill Sluta? Vill du kanske ha ett Nytt eller Börja om?

> jag vill gärna ha ett nytt recept

Willy: Det finns inga fler recept med de villkoren
Vill du Börja om eller Sluta?
> då vill jag börja om
Willy: Vad blir det den här gången?
> jag vill laga spenatsoppa
Willy:
Detta recept har jag hittat:
karlalfreds dundersoppa (spenatsoppa)
Kategori: soppor
2 portioner
Ingredienser:
500 gram spenat
1 tärning grönsaksbuljong
vatten
salt
peppar
1 klick smör
2 ägg
Beskrivning:
Smält den frusna hackade spenaten i en gryta, tillsätt en buljongtärning.
Späd med vatten till önskad konsistens.
Krydda och låt soppan koka fem minuter.
Precis innan soppan är klar tillsätts en klick smör.
Servera med kokta ägg i klyfotr.
Du kan ändra antal portioner på detta recept om du vill. Annars kan du Börja om igen eller Sluta.
> jag vill börja om
Willy: Vad blir det den här gången?
> vi är 5 personer och en är vegetarian har du något med gurka och tomat?
Willy: Ni är 5 personer. Jag tar hänsyn till vegetarian. Sak det vara nåt mer än tomat och gurka?
> just det ja vi är 4 personer och jag vill ha med ost
Willy: Ni är 4 personer. Du vill ha med ost, tomat och gurka. År det nåt som inte ska vara med?
> jag är allergisk mot ananas
Willy: Du vill ha med ost, tomat och gurka men inte ananas. Jag tar hänsyn till vegetarian.
Detta recept har jag hittat:
tacos (linstacos)
Kategori: vegetariskt
4 portioner
Ingredienser:
3.6 dl gröna linser
8.4 dl vatten
1 tärning grönsaksbuljong
1 lagerblad
1 stjälk selleri
1 vitlöksklyfta
1 påse taco seasoning mix
gurka
tomat
ost
sallad
lök
1 burk taco-sås
tacoskal
Beskrivning:
Hacka selleristjälken.
Lägg linserna i vattnet. I en någorlunda stor kastrull tillsammans med vattnet, buljongtärningen, lagerbladet och sellerin.
Pressa i vitlöken.  
Koka upp, minska värmen och låt sjuda i 1 timme.  
Rör om då och då och ha i mer vatten om det behövs.  
Ta bort lagerbladet när fyllningen har kokat färdigt.  
Värm upp tacoskalen, fyll dem med fyllning, toppning och sås.

Är du nöjd med detta recept och vill Sluta? Vill du kanske ha ett Nytt eller Börja om?

> jag vill sluta

Willy: Koka soppa på en spik!

Hejdå!

% Break level 1
% source_info,1
| ?-