UNIVERSITY OF GOTHENBURG

Project: Recipe Dialogue System

Student name: *Dominik Künkele*

Course: *Dialogue Systems 2 (LT2319)*
Due date: *November 1st, 2022*

Introduction

In this project, I wanted to create something, I could also actually use in my daily life. Since I like very much to cook, building a dialogue system that supports here, was not far fetched. The goal however was not only to create a system that can handle just a few recipes and situations, but can also be extended easily with many more recipes. For this, I wrote a script in *python* that can translate a more user friendly description of a recipe into the necessary goals, actions and so on in TDM.

APIs

I didn't use any explicit external API in the project. However, I relied heavily on the http service to add functionality to the dialogue system and look up information from local files. This includes firstly a lookup dictionary for detailed information about steps in the recipes. Here, I listed the needed ingredients for each step with their amount and form (e.g. onions: two, roughly chopped), used objects and their temperature (e.g. oven: 180 degrees) as well as information about time end ending conditions of a step. With this service, I was able to feed the system the needed information for a specific step. Was the information not specified in the current step, it utilized the information from to the last step. With this approach, I could respond to retroactive questions of a user. A simple lookup file lookes like the following:

```
"omelette_recipe": {
    "ingredients": {
        "egg": {
            "amount": "three",
            "form": null
        },
        "salt": {
            "amount": "a pinch",
            "form": null
        }
    },
    "steps": {
        "omelette_recipe_step_0": [
            {
                "time": null,
                "condition": null,
                "ingredients": {
                    "egg": {
                        "amount": "three",
                        "form": null
                    }
                },
                "objects": {
                    "bowl": {
                        "temperature": null
                    }
                }
            }
        ],
        "omelette_recipe_step_1": [
            {
                "time": null,
                "condition": null,
                "ingredients": {
                    "butter": {
                        "amount": null,
                        "form": null
                    }
                },
                "objects": {
                    "pan": {
                        "temperature": "high heat"
                    }
                }
            }
        ]
    }
}
```

Listing 1: Example of recipe loopup

Secondly, I used a simple dictionary to look up substitutes for ingredients. Unfortunately, I couldn't find an API for it, but just text on a website. To make the http service faster and simpler, I downloaded the data and stored it in a json file, which can be easily accessed by the service.

Data collection

For the data collection I recorded four dialogues. Two of them lasted around 10 minutes, while only the first minute was recorded for the other two. Still, these short beginnings held some interesting information and I distilled and analyzed them as well.

The dialogues were recorded in a roleplay scenario and not actually while cooking. This of course has a high impact on how the participants speak and real dialogues would bring more authenticity and should be included in the future.

Analysis of content. First, I tried to analyze the dialgues in respect to the domain and how the participants were talking about the topic of cooking. The instructions consisted of a verb that was either intransitive, transitive or ditransitive. The transitive and ditransitive verbs only referred to *ingredients* or *objects*. If it was an ingredient, they could add the amount (e.g. 'two teaspoons') to use or the form, it should be in (e.g. 'roughly chopped'). Objects could optionally be enriched with information about it's temperature. In some cases, the instructor could add also information to the whole step about time (e.g. 'for three minutes') or an ending condition (e.g. 'until it appears glassy').

S>: OK, perfect. And also ähh juniper berries.
U>: Hmm. I have.
S>: Hmm..ok. Around a teaspoon.
U>: Hmm

Dialogue 1: Ingredient with amount

S>: And now you can start preheat the oven for 220 degrees.
U>: I have done it. What should we do?

Dialogue 2: Object with temperature

S>: Leave the meat for further 15 minutes.
U>: OK, done.
S>: The rind should be very crusty, but not burned.
U>: Hmm, hmm.
S>: So around 15 minutes. If takes longer, leave it longer. If it is already...
U>: Should I take a look sometimes?

Dialogue 3: Step with time and condition

These attributes are also used in the *recipe_lookup* mentioned in the APIs chapter and are the core of the implemented dialogue system itself.

Analysis of dialogue. Looking at how the dialogue was built up, I could also find some interesting ideas. These can be differentiated into behaviour of the system and behaviour of the user.

Acknowledgements (user). It was very noticeable, how often the user was acknowledging and confirming the utterances by the system. I could differentiate three different types. In the first, the user was just using one word (e.g. 'ok', 'yes') or a non-lexical sound (e.g. 'hmm', 'uh-hmm'). In the second one, the user repeated a word from the previous system utterance. This can also be seen as a perceptual confirmation in some cases. In the last, the user describes the current situation and by this verifies it with the system. This type is often on the border of being an acknowledgement or being a question by the user. Dialogues 4 and 5 show some examples.

S>: Then 400 grams of mixed mushrooms.
U>: Mixed mushrooms, OK, OK, Champions, do they work? (Type 2)

Dialogue 4: Acknowledgement (Type 2)

S>: OK, you can try to feel the meat with the tip of a knife. if it's tender or not. Is it tender?
U>: It's bleeding. (Type 3)
S>: Then you can give it another 30 minutes in the oven.
U>: OK. (Type 1)
S>: And the oven goes back up to 220 degrees.
U>: Hmm (Type 1)

Dialogue 5: Acknowledgement (Type 1 and 3)

Early steps (user). Another behaviour by the user was that they were sometimes doing steps, before they were told to by the instructor. In Dialoge 6, the user is deciding to already cut the onions, while the instructor is just listing the ingredients. Later in the dialogue, the instructor remembers it and omits the actual step in the recipe.

S>: [...] Additionally you need also two onions.
U>: Yes.
S>: And they ähh need to be peeled and also cut later in...
U>: I will do that now.
S>: ...three thick slices. Not chopped.
U>: Done
S>: OK. [...]

Dialogue 6: User is doing steps before instructed

Substitutions (user). Furthermore, the user was also asking the instructor to substitute ingredients. This was done by either explicitly proposing an ingredient or just asking the instructor for any ingredient. In some cases the user was also explicitly stating, which ingredient should be substituted, in most cases, this information was implicit in the context. Dialoges 7 and 8 show some examples.

S>: Ähh, we need garlic.
U>: I have allergic to garlic.

S>: Ohh
U>: What could what should I use instead?
S>: I think you can just leave it out in this case.

Dialogue 7: User asking for substitutions

S>: Then 400 grams of mixed mushrooms.
U>: Mixed mushrooms, OK, OK, Champions, do they work?
S>: Yes, here it suggests button and Portobello.
U>: Ahh
S>: But I think it just, uh, accords to taste.

Dialogue 8: User explicitly proposing substitution

This doesn't only apply to ingredients, but also objects as seen in Dialogue 9. The instructor then also remebers that the ingredient/object were substituted and uses the substitutions in the rest of the dialogue.

S>: I know. Cover the tin with ähh with kitchen foil.
U>: I don't have a kitchen foil. What should I use instead?
S>: Then you can maybe use a lid
U>: OK.

[...]

S>: ...very quick 45 minutes. And now you take out the pork and remove they lid.

Dialogue 9: Instructor remembering substitutions

Jokes (user/system). Additionally, the user, as well as the instructor were using jokes or 'unserious' utterances multiple times. This consisted of answering in a jokingly way or also making a joke why the user or respectivley the instructor didn't say anything for some time. Dialogue 10 gives an example.

S>: [...] Stir them around a bit, not too much, to not break them.
U>: Doing it, doing it.

[*pause*]

S>: What are you doing? Are you going to collect the mushrooms youself?
U>: Haha..ok. I'm done.

Dialogue 10: Instructor making a joke after the user didn't say anything

Ask for specific action (user). In the dialogues, it could also be seen that the user sometimes asks if they should do a specific action. In contrast to a *how* question, the user already proposes something he should (or should not) do.

S>: Add the mushrooms for three to four minutes or until they are soft.
U>: OK. Hmm..ok. Throwing mushrooms in. Shall I stir them around or just let them be?
S>: Yes..yes. Stir them around a bit, not too much, to not break them.
U>: Doing it, doing it.

Dialogue 11: User asking for specific action

Giving information step by step (system). In some cases, the system was not giving all information in one step, but in multiple turns. That happened for example, when the instructor was reading the ingredients. Sometimes they were giving the ingredient in the first turn and after the user acknowledged it they gave the amount or form (see Dialogue 12).

S>: OK, perfect. And also ähh juniper berries.
U>: Hmm. I have.
S>: Hmm..ok. Around a teaspoon.

Dialogue 12: Instructor giving information in multiple turns

Wating for user (system). Most of the time the instructor was giving the next instruction, after the user acknowledged the previous. But occasionally, the instructor also waited for the user to request the next step. This may be due to the roleplay setting and would need to be verified in future experiments.

S>: and then press the cream paste into it evenly.
U>: I have done it.
S>: Hmm
U>: What’s next?
S>: Use the dill, the rest of the dill. You have the fronds

Dialogue 13: Instructor waiting for user to request step

Adapting to user language (system). In some few cases, the instructor also adpated his answers to the language and words, the user used as seen in Dialogue 14.

S>: Then you should chop the Rosemary finely.
U>: OK. 3 branches or how many was it?
S>: It was just one to two tablespoons...
U>: Tablespoon.
S>: ...of just the chopped Rosemary.

Dialogue 14: Instructor reusing words of the user

Implementation

As mentioned, I wanted to create a *generator* that is translating an easy to write description of recipes into the necessary changes of the TDM files (Ontology, Domain, Grammar, nlg, expected input, visual output). There are two main reasons for this approach.

First of all, an end-user shouldn't need to have too much knowledge of how to implement it technically. These changes furthermore need to be aligned and are not allowed to be in conflict with each other. If a user would only need to change one file with a rather simple syntax, the user experience would be much better, and errors will be prevented.

The second reason that was also much more important for myself in the last weeks is the simplification of the develeopment process itself. With this approach, I didn't need to worry of forgetting to change files or to add specific tags to a file. I only needed to define them once in the *generator* and it was applied to all recipes. Furthermore, this made it much easier to change a certain behaviour that applied to all recipes, since I only needed to change the template instead of every single recipe itself. And lastly, this also gave me a quick lookup, how I implemented a feature and which files it involved.

One restriction of this approach is that I could implement features only in a generic way, since it needs to take all recipes into account. Very specific recipe-dependent requirements couldn't be implemented in an easy way. On the other hand, this makes the system predictable and gives the user a better understanding of the features and how to use them.

Generator. The generator is adapting the content of the following files: *domain.xml*, *ontology.xml*, *grammer_eng.xml*, *nlg.json*, *visual_output.json* and *expected_input.json*. Some parts of these files are recipe-independent. These static parts can be added to the corresponding template file. The generator will then read the XML file containing the recipes (*recipes.xml*) and insert the dynamic recipe-dependent parts into the templates. Furthermore, it will generate the *recipe lookup* file that contains information about each step of all recipes.

recipes.xml. The *recipes.xml* file has the following structure:

```xml
<recipes>
    <recipe name="omelette" image_url="www.link.to/image">
        <utterances>
            <utterance>make an omelette</utterance>
            ...
        </utterances>
        <ingredients>
            <!-- can have attributes 'amount' and 'form' -->
            <ingredient name="onion" amount="two" form="roughly chopped">
```

```
                We need two onions
            </ingredient>
            ...
        </ingredients>
        <steps>
            <step>
                <!--
                    - can have general attributes 'time' and 'condition'
                    - can have attribute 'ingredient' with optionally 'amount'
↪   and 'form'
                    - can have attribute 'object' with optionally 'temperature'
                -->
                <substep ingredient="egg" amount="three" object="bowl">
                    Crack three eggs on a bowl
                </substep>
                <substep ingredient="salt" amount="a pinch">
                    whisk with a pinch of salt.
                </substep>

                <!-- optional -->
                <how>
                    <step>
                        <!-- substeps can't contain any attributes -->
                        <substep>
                            Strip the leaves off with your fingers
                        </substep>
                        ...
                    </step>
                    ...
                </how>
            </step>
            ...
        </steps>

        <finisher>Now serve and enjoy your meal</finisher>
    </recipe>
    ...
</recipes>
```

Listing 2: Example of recipes.xml

In the following paragraphs, I will include the corresponding tag of this file with an emphasized attribute in square brackets. If for example the *domain.xml* uses the *name* attribute of the *ingredient* tag, this will be referenced as [<ingredient *name*="..." />].

Domain. Every recipe will generate a *perform goal* in the *domain.xml*. Each *perfom goal* consists of two parts, the listing of the ingredients and the step-by-step instructions. The analysis of the recorded dialogue showed that the user doesn't necessarily acknowledge every ingredient, but just let's the system/instructor read all ingredients. That's why, I included a way to optionally acknowledge an ingredient. If the user doesn't say anything, the system automatically goes on with the next ingredient after a pause of three seconds. The following code is added for each ingredient:

```
<if>
    <proposition predicate="omelette_recipe_salt_read" value="false"/>
    <then>
        <inform insist="true">
            <proposition predicate="read_ingredient_list"
                         value="omelette_recipe_salt"/>
        </inform>
        <end_turn expected_passivity="3.0"/>

        <forget predicate="proposed_ingredient"/>

        <assume_shared>
            <proposition predicate="omelette_recipe_salt_read" value="true"/>
        </assume_shared>
        <assume_shared>
            <proposition predicate="which_ingredient" value="salt"/>
        </assume_shared>
    </then>
</if>
```

Listing 3: Listing of ingredients

The reason for this complicated structure is the behaviour if the `<end_turn/>` tag. When the user only acknowledged the ingredient or didn't say anything, the system worked perfectly fine. But as soon as the system switched to another goal (after the user e.g. asked a question) and returned back, it got stuck in the *expected_passivity*. For this, I introduced a *boolean* variable that kept track if an ingredient was already read to the user. If so, it would be skipped, when returning back to this goal.

Furthermore, I stored the name of the ingredient [<ingredient *name="..."* />] in the predicate *which_ingredient*. This will get useful later, when the user raises questions like "How much?" or similar without explicitly specifying the ingredient, they are talking about.

Each step of a recipe will add the following to the *goal* in the domain:

```
<assume_shared>
    <proposition predicate="current_step" value="omelette_recipe_step_0"/>
</assume_shared>
<assume_shared>
    <proposition predicate="which_ingredient" value="salt"/>
</assume_shared>
<assume_shared>
    <proposition predicate="which_object" value="bowl"/>
</assume_shared>
<get_done action="omelette_recipe_step_0"/>
```

Listing 4: Step of a recipe

The analysis showed that the user always acknowledged a step in a recipe. For this reason, I used a `<get_done/>` element. As before, I try to give the system as much information as possible about the current state. Accordingly, I fill the predicates *current_step*, *which_ingredient* and *which_object* [last substep in step: <substep *ingredient="..." object="..."*>].

If the step contains a `<how/>` element in the *recipes.xml*, the generator will create also a new *perform goal* with the name of the step. This new goal will only contain the `<get_done/>` elements and will not share any propositions. The reason for this is that the effort for implementing it seemed complex while the benefit was not to big. In the recorded dialogues, the users never asked further questions in these situations. In the future, this could be implemented to also react to the less common responses.

The static, recipe independent part of the domain consists of the questions the user can ask during the recipe. This includes question about the five core attributes as well as a request for substitutes and grounding of a mentioned ingredient. All questions are implemented as service queries in *python* and take the previously shared predicates as arguments. Optionally, the user can overwrite these in his questions.

Ontology. The ontology contains the necessary predicates for each service query as well predicates for the entities that are currently under discussion (recipe, step, ingredient, object). Furthermore, I introduced another predicate *proposed_ingredient* used for substitutions. This is utilizing the sort *proposed_ingredient_sort* instead of *ingredient*, to help *Rasa* differentiating these when a user is asking a question in the following format:

```
<item>Can I use <slot predicate="proposed_ingredient"/> instead</item>
```

Unfortunately, *Rasa* was often still recognizing an ingredient in this slot as an enitity of the normal sort *ingredient*.

The *generator* created the rest of the necessary predicates, actions and individuals.

Grammar. The static part of the grammar consists mainly of utterances, how a user can ask one of the questions. Furthermore, it also includes some utterances for system questions. Normally, theses won't be used, since the referenced predicates will already be shared in the domain.

The *generator* then adds utterances to refer to the recipes ([<utterances/>]), individuals for the listing of the recipes [<ingredients><ingredient>*utterance*</ingredient></ingredients>] and individuals for all ingredients and objects [<ingredient *name=""*/>, <substep *ingredient="" object=""*/>].

NLG. The *generator* adds utterances for all `<get_done/>` elements of the main recipes as well as the `<how/>` actions and what the system says, when the recipe is finished.

Sample dialogues

In the following dialogues, I collected some important samples which the system can handle. The interaction test file as well as the demo video contain more examples and use cases for the system.

S>: In a large mixing bowl, toss potatoes with oil, sugar, salt and red pepper.
U>: How much salt?
S>: As much as you like. Spread on baking sheet in one layer.
U>: done
S>: Bake for 30 minutes in the preheated oven, until potatoes are tender and browned.
U>: done
S>: Add some salt to the potatoes
U>: How much salt?

S>: Uhh, a tablespoon is fine.

Dialogue 15: Handling different amounts in steps

S>: Three garlic cloves, finely sliced.
U>:
S>: Then 400 grams of mixed mushrooms.
U>: ok
S>: 1-2 tablespoons of Rosemary, according to the taste. Also finely chopped.
U>: hmm, ok
S>: And 200 grams of Risotto rice.
U>: hmm
S>: Then 100ml of white wine.
U>: yes
S>: Chop the onions.
U>: Done.
S>: Then finely slices garlic.
U>: What next?

Dialogue 16: User using different acknowledgements

S>: Chop the onions.
U>: ok
S>: Then finely sliced garlic
U>: How big should the onions be?
S>: It should be eh finely chopped.
U>:
S>: Then finely sliced garlic

Dialogue 17: Retroactive questions

S>: Chop the onions.
U>: the rosemary?
S>: No, the onion.

Dialogue 18: Grounding

S>: Chop the onions.
U>: did you say onion?
S>: Yeah.

Dialogue 19: Grounding with different inflection

S>: One onion, finely chopped.
U>: can i use a squash instead?
S>: I don't think so.
U>:
S>: Three garlic cloves, finely sliced.
U>: i am allergic to that
S>: You could use minced shallots, garlic powder, granulated garlic, or garlic salt instead.

Dialogue 20: Substitutions with and without proposals

Discussion & Future work

In Table 1, I listed the features that I found during the analysis and marked if they are already implemented or not. It can be seen that there are still a lot of features that can be added to the system. In a next step, further dialogues in a real setting should be recorded and analyzed, to let the system handle more situations in a natural way.

Feature		system/user	implemented
Handling core attributes	amount	user	✓
	form	user	✓
	temperature	user	✓
	time	user	✓
	condition	user	✓
Acknowledgement	type 1	user	✓
	type 2	user	✓
	type 3	user	✗
Early steps		user	✗
Substitutions	without proposal	user	✓
	with proposal	user	✓
	with memory	system	✗
Ask for specific action		user	✗
Jokes		user/system	✗
Giving information step by step		system	✗
Waiting for user to request step		system	✗
Adapting to user language		system	✗

Table 1: Implemented features of analysis

Additionally, dialogues with machines profit from more variation of the systems utterances. This could be for example a random selection of predefined utterances. Unfortunately, this didn't work so far in my project.

Problems & feature requests

In the following, I list some problems I had during the implementation. These may also be considered to add in future versions of TDM.

default values for predicates When I used boolean predicates to indicate if an ingredient was already read to the user, it would have been helpful to set the default value. In the end, I utilized the *top* goal fot that.

comparing predicates At the moment, I didn't find a way to compare the values of two predicates. This would have been useful for example to ground a mentioned ingredient. Here, I would have liked to compare a predicate that stored what the system said, with a predicate that stored what the user said. This would also apply to comparing number or string predicates with actual values. For my project, I implemented this in python.

optional acknowledgement by the user In my project, I needed to implement a quite complex structure for this (see above), to not freeze the system after a switch to another goal and then returning back.

multiple propositions in *assume_shared* elements At the moment, I am using up to three `<assume_shared/>` elements for each step. Being able to add all propositions to one `<assume_shared/>` elements would increase the readability of the code.

variablity At the moment, the system is repeating a step instruction always in the same way. It would be nice to add some variablity to this and change, how the system repeats the instructions. This also applies to the situation, when the system returns to a goal (e.g. "chop the onions", "how many?", "three", " ", "chop the onions"). In theses cases, the system may even omit the repetition completely.

proposition with parameters For each step in the recipe, I shared, which ingredient was currently used, to react to questions by the user appropriately. I also would have liked to share the attributes *amount* and *form* for this specific ingredient. Unfortunately, I didn't find an easy way to do that. Filling a predicate *amount* would give a wrong answer if the user referred to another ingredient (e.g. from a previous step) in his question. In this situation, it would be helpful, to declare something like this:

```
<proposition predicate="which_amount" parameter="onion" value="two"/>
```

The parameter onion must then also be somehow connected to another *predicate*, maybe like this in the ontology:

```
<predicate name="which_ingredient" sort="ingredient"/>
<predicate name="which_amount" parameter="which_ingredient"
↪  sort="amount"/>
```

In my project, I utilized the service query and python for this problem.