Subway Sandwich Interactor in Prolog

Naoki Honda (N1804369J)

November, 2018

1 Introduction

This is the documentation for a simple script in SWI-Prolog that interacts with customers as a waitress of the Subway [1]. To start, enter "`order.`"
The prolog script offers different meal options (sandwich options, meat options, salad options, sauce options, top-up options, sides options etc.) to create a customized list of person's choice. The agent will intelligently select the option based on previous choices. For example, if the person chose a veggie meal, meat options will not be offered. If a customer chose healthy meal, fatty sauces will not be offered. If a customer chose vegan meal, cheese top-up will not be offered. If a person chose value meal, no toppings will be offered.

The following assumptions were observed in this implementation to make it simple:

- no abortion until it's done – This agent assume that the customer will always be there until the order is done. Since there's no implementation that deals with the abortion in case the customer changes his/her mind, only way to stop the interaction before finished is to close the window;
- sauce order has up to 2 choices – Customer can order sauce up to 2 kinds;
- multiple veggies order – The agent allows customers to order veggies as many as they want;
- no multiple order for other categories – for other orders, (meals, breads, meats, toppings, sides be specific,) the agent will not allow customers to order more than 1 choice.

The sections below describe the rationale and the details of this implementation. Section 2 shows how the agent outputs all the available options for a specific customer, taking care of the previous orders. Section 3 explains how the agent asks order and store the choice into its 'Knowledge Base'. Section 4 describes how the knowledge acquired from the customer is displayed at the end of the interaction. Finally, Section 5 explains the problems that we faced as we build the agent, and current defects for further extension.

[1] `https://www.subway.com/en-US` (accessed on November, 2018)

2 Showing available options

For this implementation, there are several categories of order and each of them has its options. Here's all the available options.

- **meals** – healthy, normal, value, vegan, veggie;
- **breads** – italian_wheat, hearty_italian, honey_oat, parmesan_oregano, multigrain, flatbread;
- **meats** – chicken, beef, ham, bacon, salmon, tuna, turkey;
- **veggies** – cucumbers, green_bell_peppers, lettuce, red_onions, tomatoes, black_olives, jalapenos, pickles;
- **toppings**:
  - **cheese** – processed_cheddar, monterey_cheddar;
  - **others** – avocado, bacon, pepperoni;
- **sauce**:
  - **fatty** – chipotle, bbq, ranch, chilli, mayo;
  - **non-fatty** – tomato, honey_mustard, sweet_onion;
- **sides** – chips, cookies, hashbrowns, soup, drinks.

2.1 Declaration of the predicates

The following predicates are to show the options:

- `index_recur`/2 (Figure 1) – is recursive predicate. Given the starting number as its first argument (when implemented, the start number is always set to be 1 since we want to show from the first one in the list) and the list we want to show as its second argument, it will show each elements with index number one by one until all the elements in the list are shown, and at the end, it will write "*Type number of your choice*" in the console;
- `index_recur_without_chosen`/2 (Figure 2) – is also defined recursively to show all the options except ones that are already chosen by the customer, given the list of chosen index numbers as its first argument. It will be used to define the next predicate of `show_options_without_chosen`;
- `show_options_without_chosen`/2 (Figure 3) – Given the list of chosen number and list of options, it will output all the options that are not selected yet. It should always be used after calling the list of options.
  (for example, "*veggies(Veg_List), show_options_without_chosen(Chosen_List, Veg_List)*", This set of code will show all veggies options without ones already chosen.).

```
% Define index_recur
index_recur(Last, List):-
    length(List, Length),
    Last is Length + 1,
    write('Type number of your choice:').

index_recur(Index, List):-
    nth1(Index, List, Elem),
    format('~w: ~w ~n', [Index, Elem]),
    Next is Index + 1,
    index_recur(Next, List).
```

Figure 1: The declaration of the predicate `index_recur`

```
% Define index_recur_without_chosen
index_recur_without_chosen([], _) :-
    write('Type number of your choice:').

index_recur_without_chosen([Head|Tail], List):-
    nth1(Head, List, Elem),
    format('~w: ~w ~n', [Head, Elem]),
    index_recur_without_chosen(Tail, List).
```

Figure 2: The declaration of the predicate `index_recur_without_chosen`

```
% Define show_options_without_chosen
show_options_without_chosen(Chosen_List, Option_List) :-
    findall(X,
(
                length(Option_List, Y),
                between(1, Y, X),
                integer(X),
                \+ member(X, Chosen_List)
)           ,
           Without_chosen),
    index_recur_without_chosen(Without_chosen, Option_List).
```

Figure 3: The declaration of the predicate `show_options_without_chosen`

2.2 Implementation

Based on the assumptions, the agent will only need to show the entire list once and no longer need to show more than once for orders of **meals**, **breads**, **meats**, **toppings**, **sides**. For those categories, we will use `index_recur` with calling each list and setting the start number of `index_recur` as 1 (Figure 4).

```
% Showing all meats options
meals(Meal_List), index_recur(1, Meal_List).
```

Figure 4: Example: showing meat options with `index_recur`

For **veggies**, we need to show not only the entire possible options at first, but also the options without chosen ones consecutively until the customer says it's done. For the first time, we will use the same exact implementation as we previously discussed, but for the second and after, we will use `index_recur_without_chosen`, and `show_options_without_chosen` and other predicates we will define in the following sections, so the actual code of implementation will be discussed in Section 3.

Same goes for **veggies**, we will discuss it in Section 3.

3 Taking order

This section describes how the agent ask order and inputs the customer's order into its "Knowledge Base", and how it then uses that knowledge to intelligently show the next option for a specific customer.

3.1 Declaration of the predicates

The following predicates are to input customer's order into the Knowledge Base:

- **`read_only`**/2 (Figure 5) – will only accept customer's input which is in the shown list. It needs the list shown to the customer as its first argument, and it returns accepted output in the second argument. It will restart from the beginning if the input is not the number in the shown list;
- **`~~_option`**/1 (Figure 6) – is the predicate to pass customer's input to the knowledge base. It requires input as an index number of the list, and with that number, it will specifies the ordered element and store it to the knowledge base.

```
% Define read_only
read_only(Specified_List, X):-
    read(Input),
(
        integer(Input)
        -> ((length(Specified_List, Y), between(1, Y, Input))
            -> (Input = X);
(        writeln('Please answer in the numbers in the list: '), read_only(Specified_List, X)));
    writeln('Please answer in the numbers in the list')
)    ,
    read_only(Specified_List, X).
```

Figure 5: The declaration of the predicate `read_only`

```
% Define all ~~~_option
meal_option(Input) :- meals(L), nth1(Input, L, Elem), assertz(selected_meal(Elem)).
bread_option(Input) :- breads(L), nth1(Input, L, Elem), assertz(selected_bread(Elem)).
meat_option(Input) :- meats(L), nth1(Input, L, Elem), assertz(selected_meat(Elem)).
veggie_option(Input) :- veggies(L), nth1(Input, L, Elem), assertz(selected_veggies(Elem)).
topping_option(Input) :- topping_cheese(L1), topping_non_cheese(L2),
                                    append(L1, L2, L), nth1(Input, L, Elem),
                                    assertz(selected_topping(Elem)).
sauce_option(Input) :- sauce_fatty(L1), sauce_non_fatty(L2),
                                    append(L1, L2, L), nth1(Input, L, Elem),
                                    assertz(selected_sauce(Elem)).
side_option(Input) :- sides(L), nth1(Input, L, Elem), assertz(selected_side(Elem)).
```

Figure 6: The declaration of the predicates `~~_option`

3.2 Implementation

3.2.1 General use

In general case, (for **meals**, **breads**, **meats**, **toppings**, **sides**) `read_only` and `~~_option` are used after showing all the option as we discussed in Section 2. In figure 7, we pick **meals** as an example.

```
% Showing all meats options
meals(Meal_List), index_recur(1, Meal_List),

% After showing all options,
% Read customer's input, but only if it is in the Meat_List
read_only(Meal_List, Meal), meal_option(Meal).
```

Figure 7: Example: showing meat options with `index_recur`

3.2.2 Veggie order

Unlike the general case, taking **veggies** order is quite complicated due to the fact that the agent needs to ask the customer as many times as the customer wants. Thus, with using `show_options_without_chosen` in Section 2, we created specific predicates only for the **veggies** order.

- **`read_only_without_chosen`**/3 (Figure 8) – is similar to the `read_only`, but the difference is that it only allows the input of index number which is in the shown list. Given the list of chosen index numbers and list of all possible choice, it will read the input and return as its third argument. To make it more general and available to use in the **sauce** order as well, we let this has the first and second argument. Therefore, it is supposed to used after calling chosen list and option list with using `findall` and possible choice list. (Example is in Figure 8);

- **`get_veg_option`**/0 (Figure 9) – will input customer's order with using `read_only_without_chosen`, and store the knowledge of order into the knowledge base;

- **`ask_next_veg`**/0 (Figure 10) – will ask whether the customer wants more veggies or not. It will restart with writing "Please answer in "y" or "n" :" in the console when the customer's input is neither "y" nor "n". When it is "y", it will restart the ordering process with executing `multiple_choice_veg` which will be explained later. If the input is "n", it will finish the **veggies** order and proceed to the next order;

- **`multiple_choice_veg`**/0 (Figure 11) – is just the combination of `get_veg_option` and `ask_next_veg` so that its definition is rather simple. However, despite its simple declaration, it plays an important role of restarting ordering process, can be seen in the declaration of `ask_next_veg`. (Figure 10) Without this predicate, the agent will fail to ask consecutive order.

```
% Define read_only_without_chosen
read_only_without_chosen(Chosen_List, Option_List, X):-
    findall(B,
(           length(Option_List, Y),
            between(1, Y, B),
            integer(B),
            \+ member(B, Chosen_List)), Without_chosen),
    read(Input),
(
        integer(Input)
                -> (
(                       member(Input, Without_chosen))
                                -> (Input = X);
(
                                writeln('Please answer in the numbers in the list: '),
                                        read_only_without_chosen(Chosen_List, Option_List, X)
)
)               ;
                writeln('Please answer in the numbers in the list'),
                read_only_without_chosen(Chosen_List, Option_List, X)
)    .

% Example use, read only the input that is not ordered yet
findall(A,selected_veggies_index(A), Chosen_List), veggies(Veg_List),
readread_only_without_chosen(Chosen_List, Veg_List, X),
% Only the accepted order will be stored in the knowledge base
veggie_option(X).
```

Figure 8: The declaration of the predicate `read_only_without_chosen` and its example use

```
% Define get_veg_option
get_veg_option:-
    findall(A,selected_veggies_index(A), Chosen_List),
    veggies(Veg_List),
    read_only_without_chosen(Chosen_List, Veg_List, Input),

    veggie_option(Input),
    assertz(selected_veggies_index(Input)),
    writeln('Do you want to add more? (Answer in \"y\" or \"n\") :').
```

Figure 9: The declaration of the predicate `get_veg_option`

```
% Define ask_next_veg
ask_next_veg:-
        read(Answer),
(
        Answer == y -> findall(A,selected_veggies_index(A), Chosen_List),
                                veggies(Veg_List),
                                show_options_without_chosen(Chosen_List, Veg_List),
                                multiple_choice_veg;
        Answer == n -> true;
        write('Please answer in \"y\" or \"n\" :'), ask_next_veg
)    .
```

Figure 10: The declaration of the predicate `ask_next_veg`

```
% Define multiple_choice_veg
multiple_choice_veg:-
    get_veg_option,
    ask_next_veg.
```

Figure 11: The declaration of the predicate `multiple_choice_veg`

3.2.3 Sauce order

Sauce order is somewhat easier than the **veggies** order because it will be up to 2 choice, but we need to make couple of predicates for this specific order.

- **ask_next_sauce_non_fat**/0 (Figure 12) – is almost the same as ask_next_veg, except chosen list and option list are derived from **sauce_non_fatty** list;
- **ask_next_sauce**/0 (Figure 13) – is just the entire sauce (fatty and non fatty) version of ask_next_sauce_non_fat.

```
% Define ask_next_sauce_non_fat
ask_next_sauce_non_fat:-
        read(Answer),
(
        Answer == y -> (findall(A,selected_sauce_index(A), Chosen_List),
                    sauce_non_fatty(Sauce_non_fatty_List),
                    show_options_without_chosen(Chosen_List, Sauce_non_fatty_List),

                    read_only_without_chosen(Chosen_List, Sauce_non_fatty_List, Sauce_non_fat_2),
                    sauce_option(Sauce_non_fat_2));
        Answer == n -> true;
        write('Please answer in \"y\" or \"n\"'), ask_next_sauce_non_fat
)    .
```

Figure 12: The declaration of the predicate ask_next_sauce_non_fat

```
% Define ask_next_sauce
ask_next_sauce:-
        read(Answer),
(
        Answer == y -> findall(A,selected_sauce_index(A), Chosen_List),
(                   sauce_fatty(Sauce_fatty_List),
                    sauce_non_fatty(Sauce_non_fatty_List),
                    append(Sauce_fatty_List, Sauce_non_fatty_List, Sauce_List),
                    show_options_without_chosen(Chosen_List, Sauce_List),

                        read_only_without_chosen(Chosen_List, Sauce_List, Sauce_2),
                        sauce_option(Sauce_2));
        Answer == n -> true;
        write('Please answer in \"y\" or \"n\"'), ask_next_sauce
)    .
```

Figure 13: The declaration of the predicate ask_next_sauce

4 Displaying the order summary

In this section, how the agent will display the order summary will be explained.

4.1 Declaration of the predicates

Displaying the order summary is the easiest part of this agent. We only need to create one simple predicate write_list.

- **write_list**/1 (Figure 14) – Given an input of a list, it will write down the elements in the list with the index number in the console.

```
% Define write_list
write_list([]).
write_list([Head|Tail]):-
(    length([Head|Tail], 1),format('~w ~n', [Head]));
(    format('~w, ', Head), write_list(Tail)).
```

Figure 14: The declaration of the predicate `write_list`

4.2 Implementation

Since the agent already have all the order in its knowledge base, with using `write_list` and couple of built-in predicates, displaying the order summary can be implemented in a quite simple way. In this way, all the order categories can be summarized so that we do not need to build some specific predicate neither. Here's the actual code (Figure 15).

```
writeln('Here\'s your order. Check if anything\'s wrong.'),

findall(A, (selected_meal(A), meals(ListA), member(A, ListA)), Order_meal),
write('Meal: '), write_list(Order_meal), nl,

findall(B, (selected_bread(B), breads(ListB), member(B, ListB)), Order_bread),
write('Bread: '), write_list(Order_bread), nl,

findall(C, (selected_meat(C), meats(ListC), member(C, ListC)), Order_meat),
write('Meat: '), write_list(Order_meat), nl,

findall(D, (selected_veggies(D), veggies(ListD), member(D, ListD)), Order_veggie),
write('Veggies: '), write_list(Order_veggie), nl,

findall(E, (selected_topping(E),
           topping_cheese(LT1), topping_non_cheese(LT2),
           append(LT1, LT2, ListE),
           member(E, ListE)), Order_topping),
write('Toppings: '), write_list(Order_topping),

findall(F, (selected_sauce(F),
           sauce_fatty(LS1), sauce_non_fatty(LS2),
           append(LS1, LS2, ListF),
           member(F, ListF)), Order_sauce),
write('Sauce: '), write_list(Order_sauce), nl,

findall(G, (selected_side(G), sides(ListG), member(G, ListG)), Order_side),
write('Side: '), write_list(Order_side), nl.
```

Figure 15: Displaying the order summary

5 Conclusion

5.1 Dealt Problems

As we construct the agent, there are couple of questions/obstacles to make it more interactable and realistic.

The following questions are the dealt obstacles as we build the agent:

- How to deal with multiple input?
- For the multiple-choice, how to show option list without the ones which already chosen?
- How to deal with customer's "false" input which is not in the list?
- How about getting customer's multiple input all at once? (Figure 16)

For the last one, we attempted to get multiple input of **veggie** order from customer all at once, and here's the predicate that we no longer use. The reason we decide not to use it is because it requires customers to input correctly. Since we don't use this predicate, customer's work of inputting gets relatively easier, but on the other hand, as its trade-off, the agent will show the list again and again until all the customer's inputs are done. It looks less realistic and a lot of work for the agent.

```
/* List input ver.*/
veggie_option_list_input([]).
veggie_option_list_input([Input_Head|Input_Tail]) :-
    veggies(List), nth1(Input_Head, List, Elem),
    assertz(selected_veggies(Elem)),
    veggie_option_list_input(Input_Tail).
```

Figure 16: The declaration of the predicate `write_list`

5.2 Defects

As we discussed, one of this agent's defect is that it can only process one order at a time. In addition, as we dealt with all the problems listed above, we found the other defect that is, with this agent, customers cannot quit ordering in the middle of the interaction. In case of further extension, we can improve this point to make it more realistic and accessible.