

**CZ3005 Assignment 3** **Report**

**Zhang Yuhan**

**U1823060F**

**SSP2**

 SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

**Overview**

The aim of this assignment is to write a prolog script to mimic the procedures of ordering a sandwich from Subway. The system will provide choices to the user in order to create a customized list from the user’s choices. The guidelines of the assignment stated that:

* The script should offer different meal options, sandwich options, meat options, salad options, sauce options, top-up options, sides options etc.
* The options should be intelligently selected based on previous choices.

**Implementations of Code**

Structure of code:

1. Create predicates for all options in different categories of choices.
2. Create dynamic predicates for storing user choices.
3. Run ‘begin\_order.’ and display a welcoming message and prompt user for a meal type.
4. Depending on the chosen meal type, display the options different categories of choices and prompt for user input.
5. After order is finished, display all user’s choices and an ending message.
6. Flush out all elements in dynamic predicates so future runs of the code will not be affected (in case the program is not restarted).

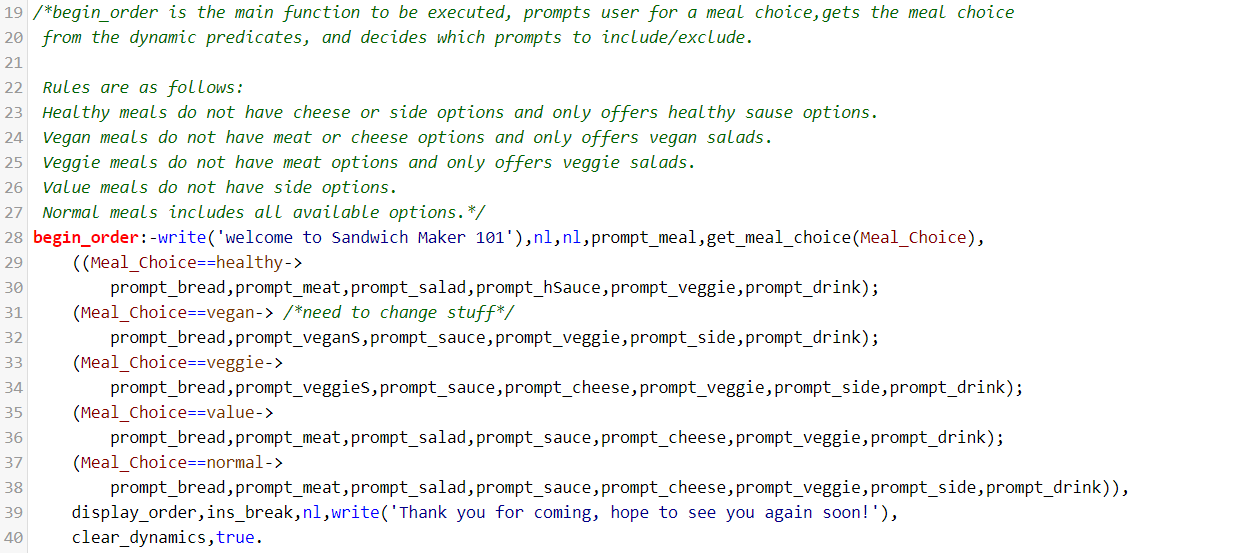


Figure 1: General flow of the program.

**1, 2.**

The following shows how predicates of options in different categories and dynamic predicates are instantiated.

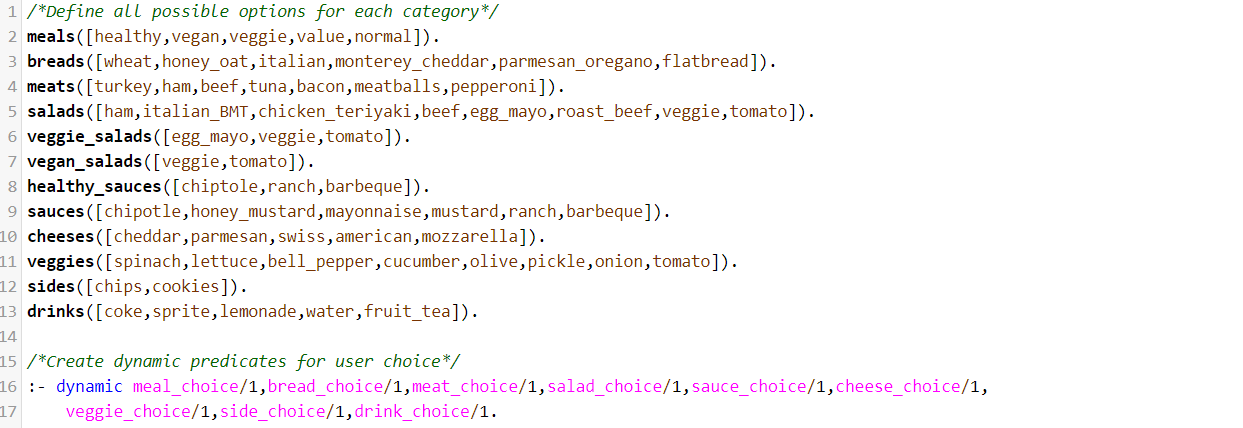


Figure 2: Instantiation of predicates

**3.**

As shown in Figure 1, while executing begin\_order, prompt\_meal is first called. This function is responsible for getting an input from the user and asserting it into its corresponding dynamic predicate, which in this case would be ‘meal\_choice’. This is also true for all other prompt functions in the program as shown below. For simplicity’s sake, I will be using prompt\_meal as an example that represents all the prompt functions. The guide and ins\_break functions only exist to make the interface cleaner to the user and does not serve any practical purpose.

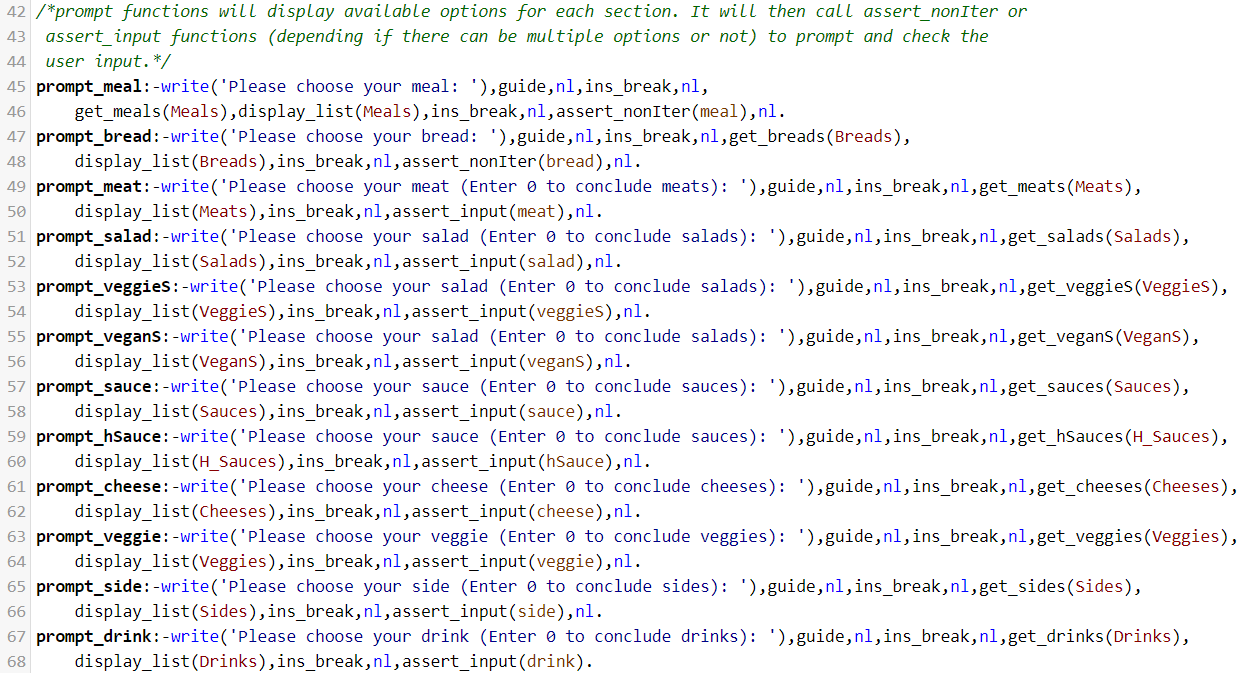


Figure 3: Implementation of ‘prompt’ functions

As can be seen from Figure 3, ‘prompt\_meal’ first needs to call the function ‘get\_meals(X)’ in order to get the list of available options the user can choose from. The predicate findall(Template,Goal,Bag) is used extensively. The logic for the ‘get’ functions is shown below.

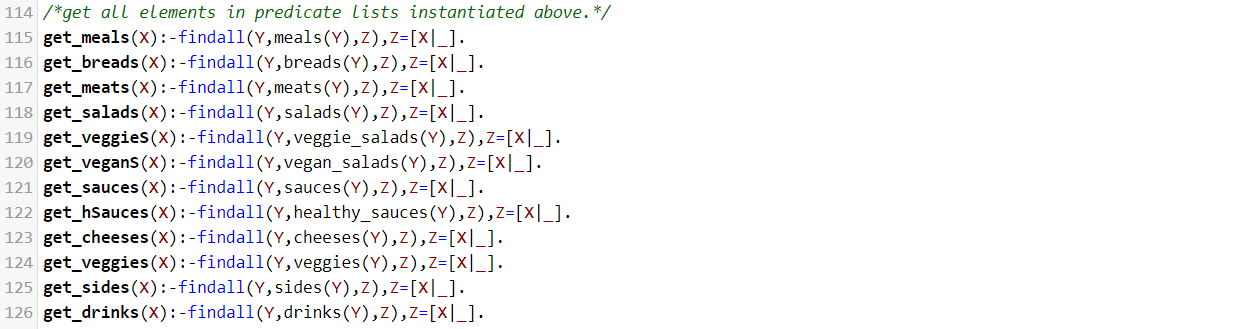


Figure 4: Implementation of ‘get’ functions

After getting the list of available meal options, ‘display\_list(X)’ is executed in order to display the options to the user. The ‘display\_list(X)’ is a recursive function that removes and prints the first element of a list, then calls itself with the remaining the list. This continues until there are no more elements left in the list, then ‘display\_list([])’. will simply return.

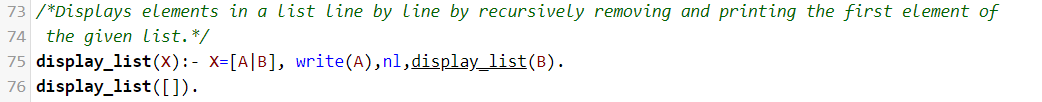


Figure 5: Implementation of ‘display\_list(X)’

After all options are displayed, either ‘assert\_nonIter(Type)’ or ‘assert\_input(Type)’ is called depending on the type/category of the desired input. For example, the input type for ‘prompt\_meal’ is ‘meal’ and there should be only one input for meal choice per order, thus ‘assert\_nonIter(meal)’ is called. On the other hand, the input type for ‘prompt\_meat’ is ‘meat’ and there can be multiple (or zero) selections of meat per order, thus ‘assert\_input(meat)’ is called.

Both functions implement similar logic where they first read the user’s input, then checks if the input matches any elements in the predicate list corresponding to the input’s ‘Type’. If there is a match, the user’s choice will be displayed as verification (‘display\_choice(Input)’ function) and the input will be asserted into the corresponding dynamic predicate. If the user’s input does not match (invalid input), the function will display an error message and call itself again to repeat the prompt until the user enters a valid input.

The main difference between ‘assert\_nonIter(Type)’ and ‘assert\_input(Type)’ is that ‘assert\_nonIter(Type)’ will stop prompting users for an input after the first valid input is found and return true while ‘assert\_input(Type)’ will continue to prompt the user for input as long as the user does not enter ‘0’.

The logic for these functions is shown below.

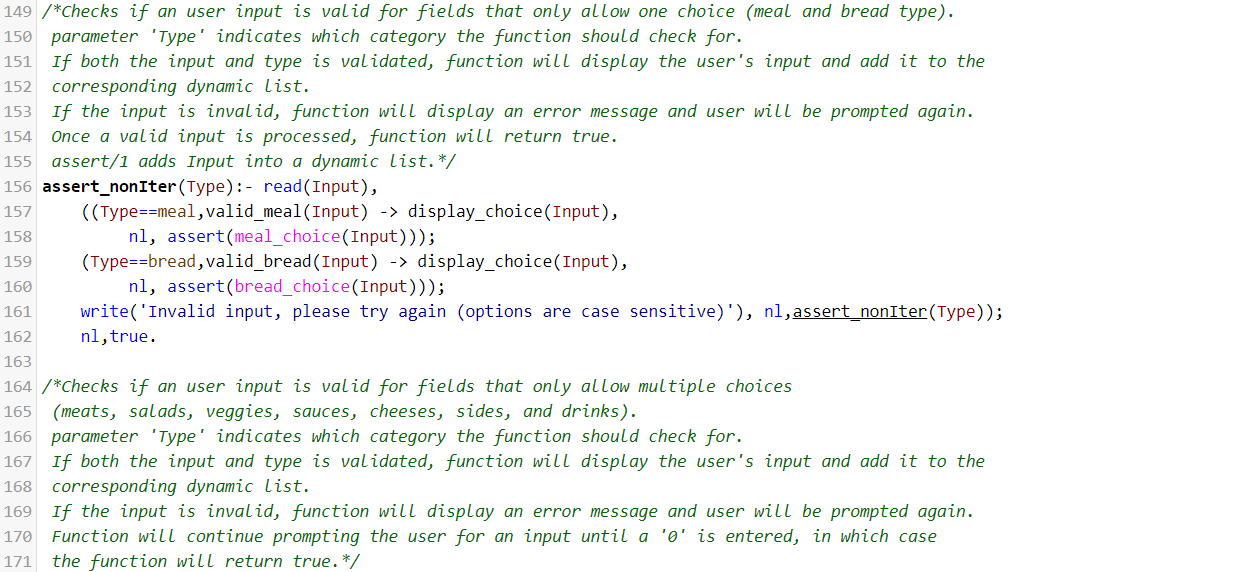
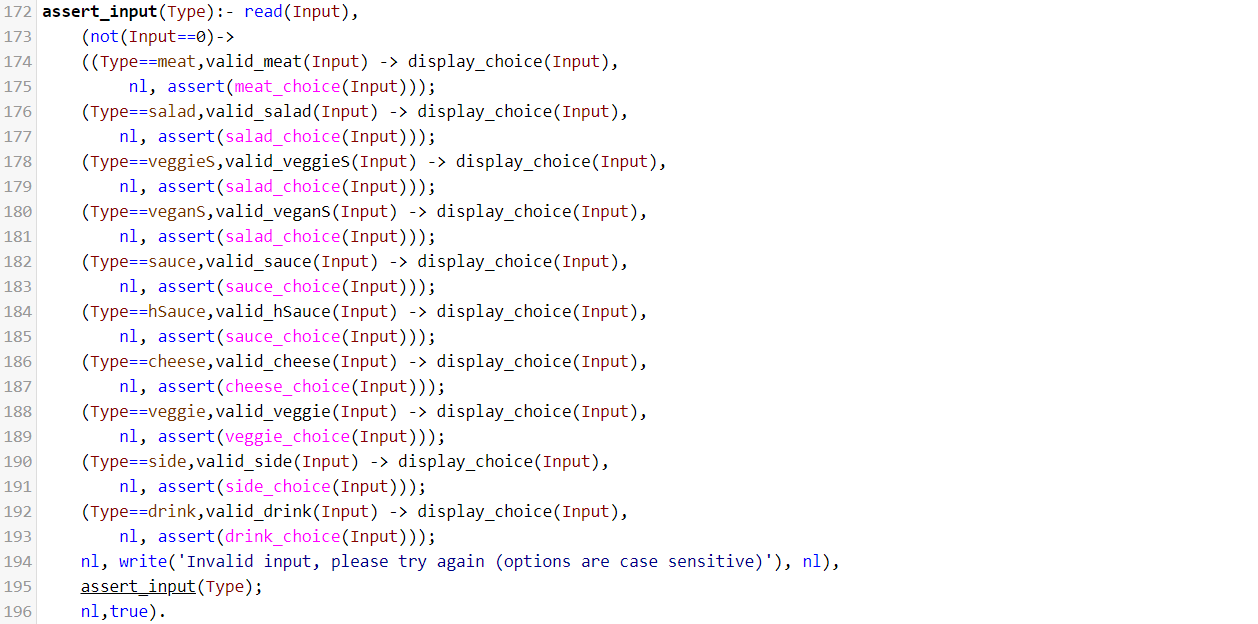


Figure 6: Implementation of ‘assert\_nonIter(Type)’ and ‘assert\_input(Type)’

As can be seen in Figure 6, the user input must be validated against the available options before it can be asserted into the corresponding dynamic predicate. This is done by calling the ‘valid\_meat(X)’ function where it gets the list of available meals and compares each element again the input to see if it is a member of the list. The predicate member(element, List) is used to achieve this result.

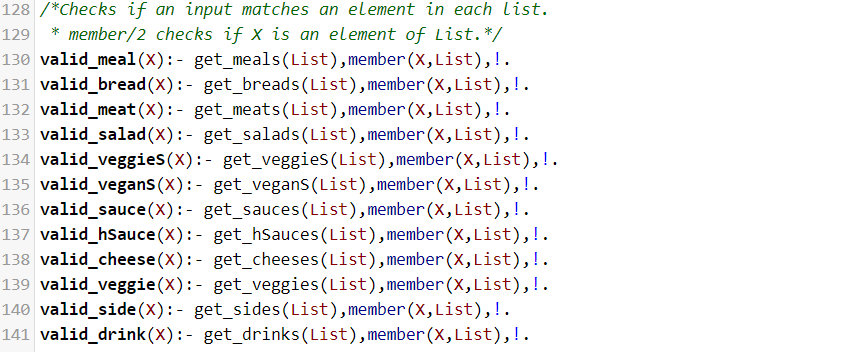


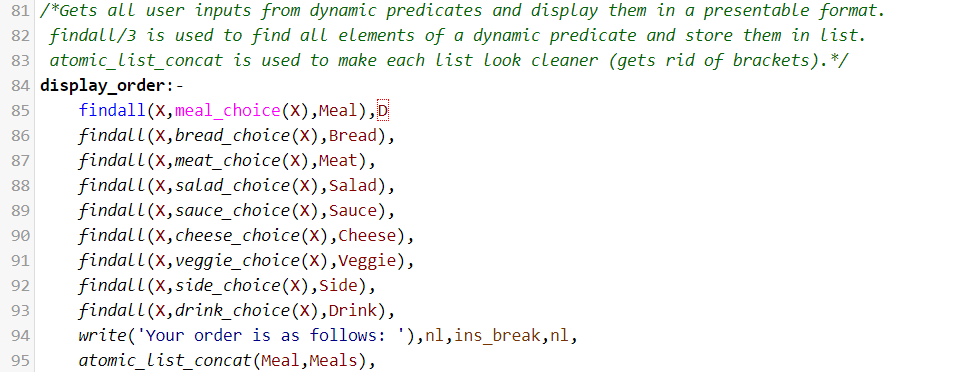
Figure 7: Implementation of ‘valid’ functions

**4.**

Depending on the user’s input for meal type, the system should include and omit different categories to prompt the user. For example, vegans should not be prompted to enter meat or cheese options and the salad options should only include those that conform with the diet of vegans. The implementation of this can be observed from Figure 1.

**5.**

After the system has obtain all necessary inputs from the user, it will display the order details to the user. This is done by calling ‘display\_order’ predicate where ‘findall(Template,Goal,Bag)’ is used to get the elements in each dynamic predicate. The lists will then be passed through the ‘atomic\_list\_concat(List,Separator,Atom)’ and printed in the terminal in order for the output to look cleaner. The implementation for ‘display\_order’ is shown below.



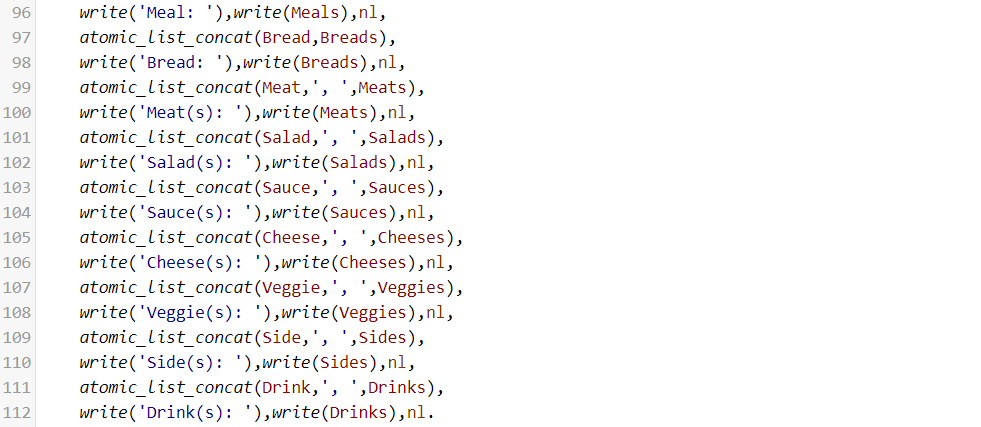


Figure 8: Implementation of ‘display\_order’

**6.**

After the user’s order and ending messages are displayed, the dynamic predicates must be flush so that future orders will not be affected. This is shown in the function ‘clear\_dynamics’ in Figure 1. Not flushing the dynamic predicates means that the inputs from the previous order will remain in the dynamic predicates and will not be overwritten by future inputs given that the program has not been restarted. This will result in the wrong lists of user input and may affect other functionalities of the program (such as getting the meal type in ‘begin\_order’). The implementation of clear\_dynamics is shown below.

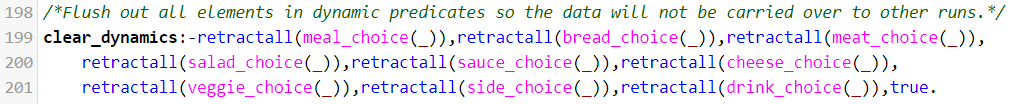
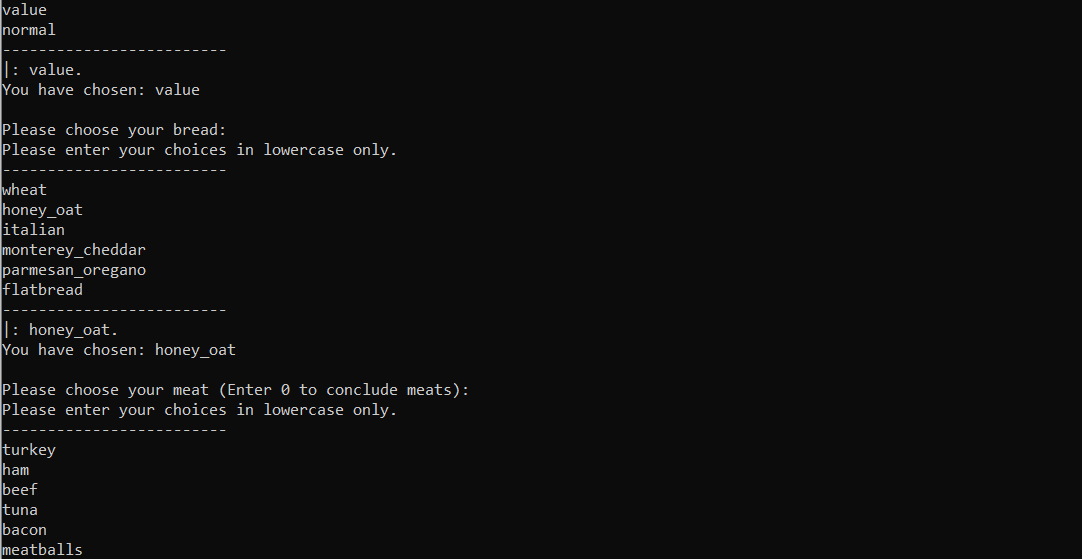
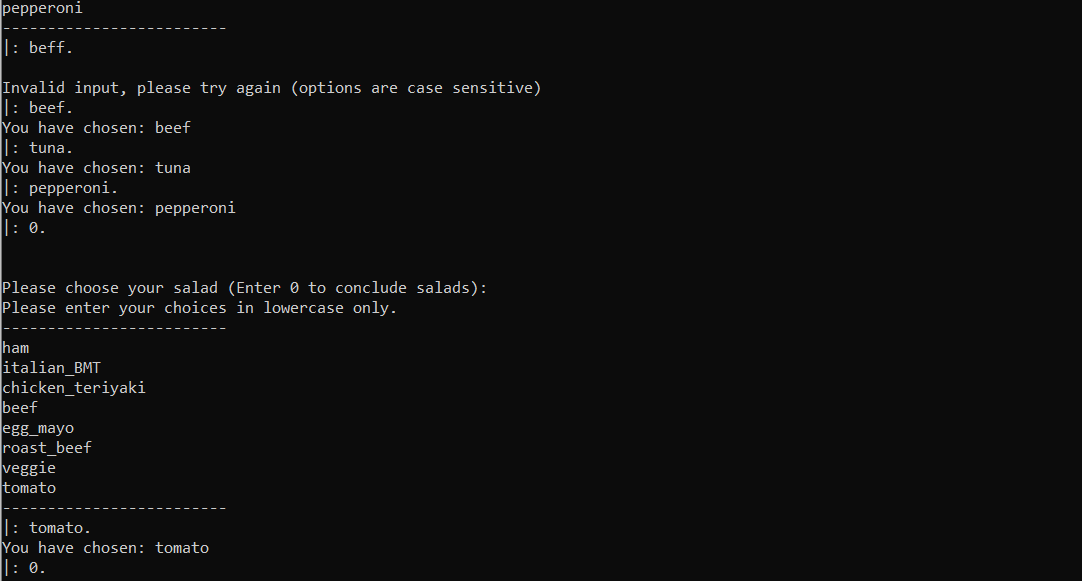
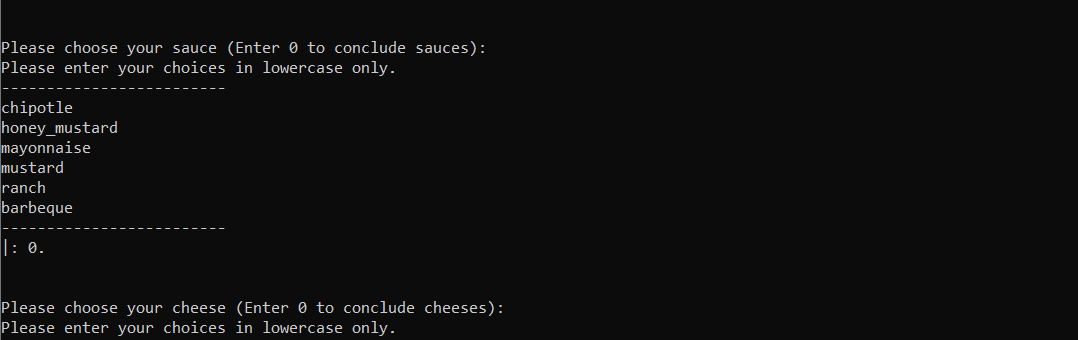


Figure 9: Implementation of ‘clear\_dynamics’

**Sample Outputs of Program**

Figure 10 below shows the outputs of a run of the ‘sandwich\_interactor.pl’. In this run, the flow of the program can be clearly observed. Cases of invalid input that requires error handling as well as the system’s response to them is also shown.





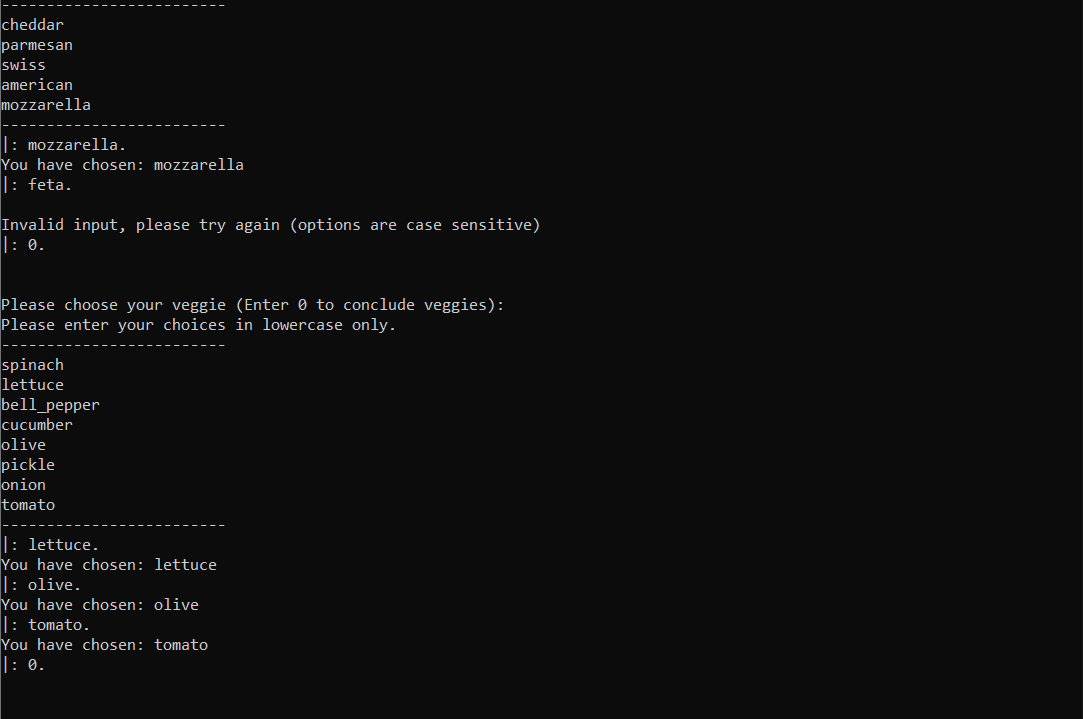
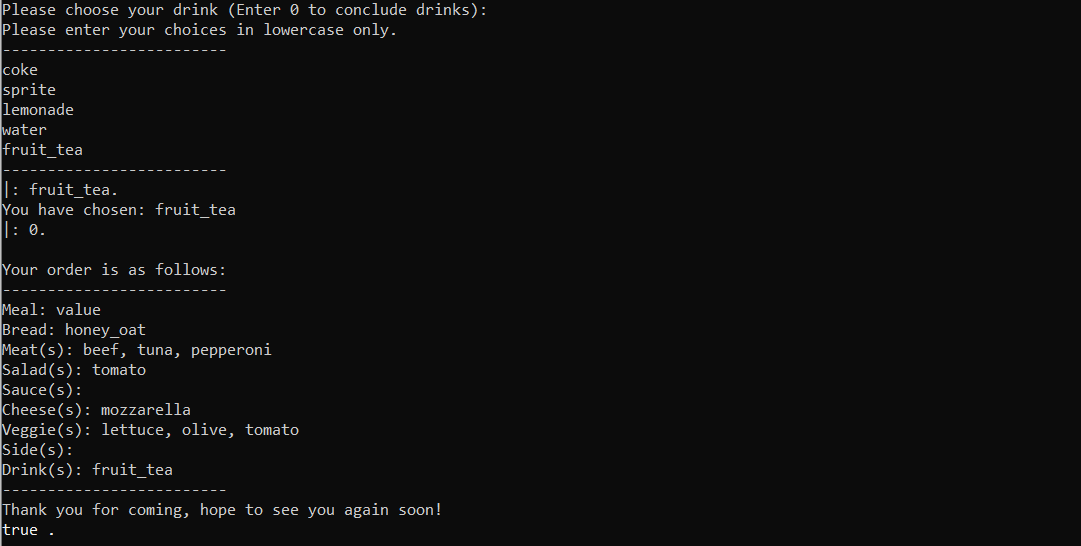


Figure 10: Prolog output from running ‘begin\_order.’