AI Lab 3 – a Subway sandwich interactor using Prolog

Tristan Lawson
N1904070A
CZ3005, TS4

Introduction

This project uses Prolog to collect a Subway sandwich order. Each user will receive a tailored experience – they have the option to choose a healthy, value, vegetarian or vegan meal. This allows the talkbot to respectfully help the user through the rest of their order.

First, the user will choose what type of order: sandwich, combo, or other. Then they can choose their meal "option", be it healthy, vegetarian, or none. The talkbot will use these responses to guide future questions. A vegan combo will ask for sandwich toppings, sides, and a drink, without taunting the user with non-vegan options such as cheese, milk, or cookies. A value sandwich will automatically limit to one of each topping and a 6-inch size for a discounted price. A healthy meal will not limit the options, but it will suggest the healthiest choice to the user.

Here is an example of the program handling an order:

```
?- order().
What can I get for you today?
  sandwich  combo  other
|: other
Would you like a special meal?
  healthy  value  vegetarian  vegan  none
|: healthy
What sides would you like?
  cookie  apple  chips  none
healthiest choice: apple
|: apple
Would you like any more?
  cookie  chips  none
|: none
Please choose a drink:
  water  milk  coke  applejuice  none
healthiest choice: water
|: water
Your healthy meal:
---------------
|apple
|water
---------------
Thank you for choosing Subway!
true .
```

Figure 1: talkbot handling a healthy order of a side and a drink

The program is designed to be executed in the Prolog query system, but it can be adapted to other platforms by changing the i/o functions. This would allow for a more beautiful user interface with buttons and pictures.

order()*ing* a meal

The program is executed by typing *order().* as a query. This prompts the talkbot to gather all necessary information to complete the user's order. Figure 2 displays the code: first the talkbot collects the meal type and option. Then it uses this information to start the order. Once it has collected the complete order, it proceeds to the checkout screen and displays the order.

```
order():-
        chooseMealType(Type),          %sandwich, combo, other
        chooseMealOption(Option),      %healthy, value, etc.
        (   sandwich(Type),chosen(sauce,Meal,Option);
            combo(Type),chosen(drink,Meal,Type,Option);
            other(Type),chosen(drink,Meal,Type,Option)),
        checkout(Meal,Type,Option).
```

Figure 2: the order() command prompts the talkbot to handle a user's complete order, then displays it in a checkout screen

The *chosen()* terms form the structure of the order. There are 8 chosen() terms, one for each aspect of the order (ex. chosen(drink,Meal,Type,Option)). Once all relevant terms are satisfied, the order is complete. As seen in the figure, the *chosen()* terms are connected by backwards chaining. The top of this chain is at the *chosen(bread,…)* term, or the *chosen(side,…)* term. This nuance is determined by the OR branch (circled in red). Its purpose is to prevent from asking a customer for sandwich toppings when they did not ask for a sandwich.

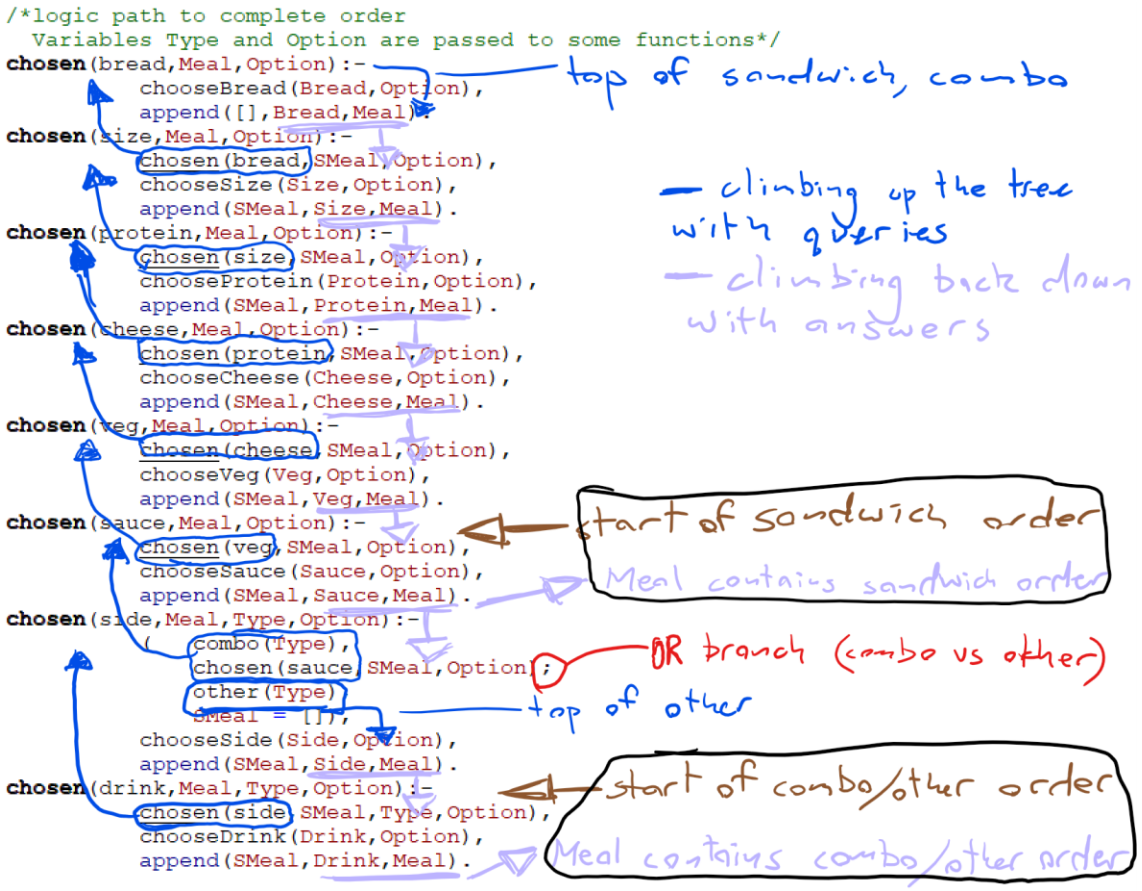

Figure 3: flow diagram for order() query. Blue shows backwards chain of queries, purple shows resolved queries.

User I/O

Talkbots are heavily reliant on user input and output. A poor user interface will make customers frustrated, no matter how good the rest of the system is. As such, it is important to develop a user-friendly interface that is easy to learn.

getUserInput()

This program is designed for Prolog, so it uses a simple prompt-and-respond setup. A question is displayed for the user, and they are asked to type their response. Each question is accompanied by a list of possible answers. If the user types an invalid response, they will be asked again.

Figure 4 shows the code for *getUserInput().* If the input *String* matches an element of the list *List*, then it is returned as *ValidString*. Otherwise, the program prompts the user to re-enter their response, and it recursively calls itself.

```
getUserInput(ValidString,List):-
          read_line_to_string(user_input,String),
          (   member(String,List),ValidString = String;
              format("This is not an option. Please try again:\n"),
              getUserInput(ValidString,List)).
```

Figure 4: getUserInput() checks if the user input matches one of the options in List*, else it prompts the user to try again and it recurses. Valid responses are returned as a "string"*

The response must exactly match one of the listed options. As you can see in Figure 5, input is case sensitive. The first attempt had a space directly after "option", which also was not accepted. This issue can be worked around by having lists of 'valid' responses that are all accepted. This is shown in Figure 6, on page 4. While it improves the customer experience, it is out of the scope of this report.

```
?- order().
What can I get for you today?
  sandwich  combo  other
|: other
This is not an option. Please try again:
|: Sandwich
This is not an option. Please try again:
|: COMBO
This is not an option. Please try again:
|: combo
Would you like a special meal?
  healthy  value  vegetarian  vegan  none
|:
```

Figure 5: example of valid and invalid user input. The program asks the user again if they do not provide a valid answer. Input is case-sensitive.

One can abort the program by clicking Run->Interrupt and typing 'a' to abort. This program does not contain an 'exit' option, though this would be quite beneficial to frustrated customers. Figure 6 samples an 'exit' option, but it is not implemented in the rest of the program.

```
/*advanced user input:
    accepts a range of inputs for each option
    and has an 'exit' input*/
advancedUserInput(ValidString):-
          read_line_to_string(user_input,S),
          (   member(S,["sandwich","Sandwich","SANDWICH"]),
              ValidString = "sandwich";
          member(S,["combo","Combo","COMBO"]),
              ValidString = "combo";
          member(S,["other","Other","OTHER"]),
              ValidString = "other";
          member(S,["x","X"]),
              ValidString = "X";
          format("This is not an option. Please try again: (type 'X' to exit)\n"),
              advancedUserInput(ValidString)).
```

Figure 6: example of advanced user input. Accepts multiple inputs for each option and provides an 'exit' function.

This user input function was not implemented, but it is an example of how to improve the customer experience. For example, “combo”, “Combo”, and “COMBO” are all accepted as input for the combo option, whereas only “combo” is valid in the *userInput()* function.

In addition, the user may type “X” if they want to quit. This is considered a valid input, so it will be returned as ValidString. The function that called *advancedUserInput()* can now use this information to clean up and quit the program. Alternatively, halt() or exit() could be thrown for a quick and simple exit.

getRecursiveInput() when single input is not enough

One of the best aspects of subway is the unlimited vegetable toppings. To translate this into code, the user is asked if they would like more toppings until they specify “none”, or they have already taken them all. *getRecursiveInput()* is used for vegetables, sauces, and sides, to allow the user to create deluxe sandwiches.

```
/*Choice is a list of strings ["s1","s2",...]
  It calls recursiveChoice, which asks user for input
  until the user types "none" or is out of options.*/
getRecursiveInput(Choice,L):-
          getUserInput(Input,L),
          (   Input = "none",Choice = ["none"];
          takeout(Input,L,SL),
              recursiveChoice(More,SL),
              append([Input],More,Choice)).
recursiveChoice([],["none"]).
recursiveChoice(Choice,L):-
          format("Would you like any more?\n"),
          options(L),
          getUserInput(Input,L),
          (   Input = "none",Choice = [];
          takeout(Input,L,SL),
              recursiveChoice(More,SL),
              append([Input],More,Choice)).
```

Figure 7: getRecursiveInput() collects a list of toppings from the user. The list Choice *is a list of strings.*

This is a much more complex function: in fact, the first function *getRecursiveInput()* calls a different function when asking for second or third toppings. This is to prevent from asking the customer the same question twice, and it allows for nuances between the first choice and subsequent choices. For example, *getRecursiveInput()* returns the input “none” as [“none”] to indicate that no toppings from the list were chosen. However, *recursiveChoice()* returns the input “none” as an empty list to be appended to the other topping choices.

```
?- chooseVeg(Veg,none).
Would you like some vegetables?
  lettuce  spinach  tomato  onion  pepper  jalapeno  none
|: lettuce
Would you like any more?
  spinach  tomato  onion  pepper  jalapeno  none
|: jalapeno
Would you like any more?
  spinach  tomato  onion  pepper  none
|: pepper
Would you like any more?
  spinach  tomato  onion  none
|: none
Veg = ["lettuce", "jalapeno", "pepper"] .

?- chooseSide(Side,none).
What sides would you like?
  cookie  apple  chips  none
|: cookie
Would you like any more?
  apple  chips  none
|: apple
Would you like any more?
  chips  none
|: chips
Side = ["cookie", "apple", "chips"] .
```

Figure 8: interaction with the getRecursiveInput() function

Two examples are shown in Figure 8. First, *chooseVeg()* prompts the user to pick their vegetable toppings for their sandwich. Each time they pick a topping, they are asked if they would like to take another from a shortened list of toppings. When the user picks "none", their list is compiled and returned. Note that the selected topping is not displayed as an option the next time around. This is because the *takeout()* function is used to remove it from the list of options for the next recursion.

The second example terminates when the user has picked every side available. They are automatically moved to the next question.

Recursive functions are more likely to trap the user in infinite loops in the case of a bug. This is a part that could greatly benefit from an 'exit' function.

Tying everything together

Now let us look at how the main logic structure interacts with the user I/O.

Each *chosen()* term calls a function such as *chooseVeg()*. As an example, *chosen(veg…)* and *chooseVeg()* are displayed together in Figure 9. *chosen(veg…)* calls *chooseVeg()*, which in turn calls a user input function.

```
chosen(veg,Meal,Option):-
          chosen(cheese,SMeal,Option),
          chooseVeg(Veg,Option),
          append(SMeal,Veg,Meal).
/*choose as many as you want (except for value meal)*/
chooseVeg(Veg,Option):-
          format("Would you like some vegetables?\n"),
          vegList(L),
          options(L),
          healthyOption(Option,L),
          (   value(Option),
              getUserInput(Choice,L),
              Veg = [Choice];
          getRecursiveInput(Veg,L)).
```

Figure 9: chosen(veg,Meal,Option) and chooseVeg(Veg,Option). chooseVeg() calls getUserInput() OR getRecursiveInput(), depending on Option.

Most of the time, *chooseVeg()* will call *getRecursiveInput()* to receive a list of vegetable toppings. The variable *Veg* is then assigned to the list.

If the customer has chosen a value meal, they are only asked for one vegetable topping. Then *chooseVeg()* calls *getUserInput()* instead. The line *Veg = [Choice]* is necessary for *Veg* to be assigned to a list of strings, not a string. (recall that *getUserInput()* returns a string)

Veg is then returned to *chosen(veg…)* and appended to the list *Meal*. At this point, *Meal* contains the customer's bread, protein, and cheese choices. After *append()* is executed, *chosen(veg…)* will be true, and *chosen(sauce…)* will continue to execute. In the case of a sandwich order, *append(SMeal,Sauce,Meal)* will complete the order, and the program will *checkout()* with all the information required.

Meal Options

There are 5 meal options. As you have seen so far, they each have implications on how the meal order is collected. Vegans and vegetarians have dietary restrictions, so the talkbot is designed to only show them options they can eat. For example, a vegan order automatically chooses "none" for cheese. This logical term is shown in Figure 10.

```
/*skips vegans*/
chooseCheese(Cheese,Option):-
          vegan(Option),Cheese = ["none"];
          format("And would you like some cheese?\n"),
          cheeseList(L),
          options(L),
          healthyOption(Option,L),
          getUserInput(Choice,L),
          Cheese = [Choice].
```

Figure 10: chooseCheese(Cheese,Option) automatically chooses Cheese = "none" if the customer selected a vegan meal

Figure 11 displays vegetarian and vegan shortlists of options. If a customer selects a vegetarian or vegan meal, they will be given option lists that have all meat (and dairy/egg) options removed. This difference is displayed by the sample interaction in Figure 12.

```
sideList(["cookie","apple","chips","none"]).
veganSideList(["apple","chips","none"]).
drinkList(["water","milk","coke","applejuice","none"]).
veganDrinkList(["water","coke","applejuice","none"]).
```

Figure 11: option lists and their vegan alternatives

```
?- chooseDrink(D,none).
Please choose a drink:
  water  milk  coke  applejuice  none
|: milk
D = ["milk"] .

?- chooseDrink(D,vegan).
Please choose a drink:
  water  coke  applejuice  none
|: water
D = ["water"] .
```

Figure 12: regular meal protein choice versus vegan protein choice

Value meals provide limited additions for a discounted price. If a customer is choosing their vegetable toppings or their sides, they will only be able to pick one. This was shown in Figure 9. Similarly, only 6-inch subs are part of the value meal. Thus, Size = "sixinch" is inferred from the choice of a value meal.

Healthy options do not restrict the menu, but the talkbot will recommend the best choice for a healthy meal. For example, when choosing your bread it will recommend "ninegrain", because whole grains are healthier than breads made of all-purpose flour.

Figure 13 displays all the "healthiest options" and the lookup table. When *healthyOption(Option,List)* is called, it first checks if the customer ordered a healthy meal. If so, it will lookup the healthiest option in the provided *List* and print recommend it to the user. If *Option* does not equal *healthy*, then the function does nothing. Figure 14 shows a customer picking a *healthy* meal. Notice that it still allows all choices.

```
/*healthy option: displays the healthiest item in the list*/
healthiestBread("ninegrain").
healthiestProtein("tofu").
healthiestCheese("none").
healthiestVeg("spinach").
healthiestSauce("none").
healthiestSide("apple").
healthiestDrink("water").

healthyOption(Option,List):-
          (   healthy(Option),
              (   breadList(List),  healthiestBread(Healthy);
                  proteinList(List),healthiestProtein(Healthy);
                  cheeseList(List), healthiestCheese(Healthy);
                  vegList(List),    healthiestVeg(Healthy);
                  sauceList(List),  healthiestSauce(Healthy);
                  sideList(List),   healthiestSide(Healthy);
                  drinkList(List),  healthiestDrink(Healthy)),
              format("healthiest choice: "),
              write(Healthy),
              format("\n"));
          (   not(healthy(option))).
```

Figure 13: healthiest options and lookup function

```
Would you like a special meal?
  healthy  value  vegetarian  vegan  none
|: healthy
Which bread do you want?
  white  ninegrain  flatbread
healthiest choice: ninegrain
|: ninegrain
What size of bun?
```

Figure 14: a customer selecting their sandwich with the healthy *option*

In conclusion

This program is a good starting point in creating a Subway talkbot. The final product is completely functional and provides a targeted experience to each customer. However, it is severely lacking in aesthetic and i/o capability. Improvements for the user input were discussed on page 4 and proposed in Figure 6, however they exceeded the scope of this project. Additional improvements would be focused around improving the prompts to the user. For example, change "none" to "no" when the customer is asked "Do you want addition toppings". There could also be a cost calculation in the checkout phase.

Thank you for reading this project. Best regards and stay safe.

```
/*meal types*/
mealTypeList(["sandwich","combo","other"]).
sandwich(sandwich).
combo(combo).
other(meal).
/*meal options*/
mealOptionList(["healthy","value","vegetarian","vegan","none"]).
healthy(healthy).
value(value).
vegetarian(vegetarian).
/*vegan implies vegetarian*/
vegetarian(Option):-
        vegan(Option).
vegan(vegan).
/*item lists*/
sizes(["sixinch","footlong"]).
breadList(["white","ninegrain","flatbread"]).
proteinList(["ham","chicken","tofu","none"]).
vProteinList(["tofu","none"]).
cheeseList(["cheddar","swiss","parmesan","none"]).
vegList(["lettuce","spinach","tomato","onion","pepper","jalapeno","none"]).
sauceList(["mustard","hotsauce","mayo","none"]).
sideList(["cookie","apple","chips","none"]).
veganSideList(["apple","chips","none"]).
drinkList(["water","milk","coke","applejuice","none"]).
veganDrinkList(["water","coke","applejuice","none"]).
/*healthy option: displays the healthiest item in the list*/
healthiestBread("ninegrain").
healthiestProtein("tofu").
healthiestCheese("none").
healthiestVeg("spinach").
healthiestSauce("none").
healthiestSide("apple").
healthiestDrink("water").
/*healthiest option lookup*/
healthyOption(Option,List):-
        (   healthy(Option),
            (   breadList(List),   healthiestBread(Healthy);
                proteinList(List),healthiestProtein(Healthy);
                cheeseList(List), healthiestCheese(Healthy);
                vegList(List),    healthiestVeg(Healthy);
                sauceList(List),  healthiestSauce(Healthy);
                sideList(List),   healthiestSide(Healthy);
                drinkList(List),  healthiestDrink(Healthy)),
            format("healthiest choice: "),
            write(Healthy),
            format("\n"));
        (   not(healthy(option))).
/*main function, asks user for meal type, meal option.
  Begins their order based on the type and option
   (ex. vegans aren't asked if they want cheese)
  After completing order, proceeds to 'checkout'*/
order():-
        chooseMealType(Type),        %sandwich, combo, other
        chooseMealOption(Option),    %healthy, value, etc.
        (   sandwich(Type),chosen(sauce,Meal,Option);
            combo(Type),chosen(drink,Meal,Type,Option);
            other(Type),chosen(drink,Meal,Type,Option)),
        checkout(Meal,Type,Option).
/*logic path to complete order
  Variables Type and Option are passed to some functions*/
chosen(bread,Meal,Option):-
        chooseBread(Bread,Option),
        append([],Bread,Meal).
chosen(size,Meal,Option):-
        chosen(bread,SMeal,Option),
        chooseSize(Size,Option),
        append(SMeal,Size,Meal).
chosen(protein,Meal,Option):-
        chosen(size,SMeal,Option),
        chooseProtein(Protein,Option),
        append(SMeal,Protein,Meal).
chosen(cheese,Meal,Option):-
        chosen(protein,SMeal,Option),
        chooseCheese(Cheese,Option),
        append(SMeal,Cheese,Meal).
chosen(veg,Meal,Option):-
        chosen(cheese,SMeal,Option),
        chooseVeg(Veg,Option),
        append(SMeal,Veg,Meal).
chosen(sauce,Meal,Option):-
        chosen(veg,SMeal,Option),
        chooseSauce(Sauce,Option),
        append(SMeal,Sauce,Meal).
```

```
chosen(side,Meal,Type,Option):-
          (   combo(Type),
              chosen(sauce,SMeal,Option);
              other(Type),
              SMeal = []),
          chooseSide(Side,Option),
          append(SMeal,Side,Meal).
chosen(drink,Meal,Type,Option):-
          chosen(side,SMeal,Type,Option),
          chooseDrink(Drink,Option),
          append(SMeal,Drink,Meal).
/*functions*/
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]):-takeout(X,R,S).
/*prints list L with space between each element*/
options([]):-format("\n").
options(L):-tab(2),takeout(F,L,SL),write(F),options(SL).
/*User must choose a meal type.*/
chooseMealType(Type):-
          format("What can I get for you today?\n"),
          mealTypeList(L),
          options(L),
          getUserInput(Input,L),
          (   Input = "sandwich",Type = sandwich;
              Input = "combo",   Type = combo;
              Input = "other",   Type = meal).
/*User has the opportunity to choose a meal option.
  vegetarian and vegan remove non-veg/vegan items from the menu
  value gives a discount meal, but with more restrictions
     (such as 6-inch sub, only 1 sauce and veg)
  healthy gives a recommendation about the healthiest choices*/
chooseMealOption(Option):-
          format("Would you like a special meal?\n"),
          mealOptionList(L),
          options(L),
          getUserInput(Input,L),
          (   Input = "healthy",   Option = healthy;
              Input = "value",     Option = value;
              Input = "vegetarian",Option = vegetarian;
              Input = "vegan",     Option = vegan;
              Input = "none",      Option = subway).
/*3 options, user must choose 1*/

chooseBread(Bread,Option):-
          format("Which bread do you want?\n"),
          breadList(L),
          options(L),
          healthyOption(Option,L),
          getUserInput(Choice,L),
          Bread = [Choice].
/*2 sizes, value meal automatically gets 6-inch*/
chooseSize(Size,Option):-
          (   value(Option),Size = ["sixinch"];
              format("What size of bun?\n"),
              sizes(L),
              options(L),
              getUserInput(Choice,L),
              Size = [Choice]).
/*limited choice for vegetarians/vegans*/
chooseProtein(Protein,Option):-
          format("What protein would you like?\n"),
          (   vegetarian(Option),vProteinList(L);
              proteinList(L)),
          options(L),
          healthyOption(Option,L),
          getUserInput(Choice,L),
          Protein = [Choice].
/*skips vegans*/
chooseCheese(Cheese,Option):-
          vegan(Option),Cheese = ["none"];
          format("And would you like some cheese?\n"),
          cheeseList(L),
          options(L),
          healthyOption(Option,L),
          getUserInput(Choice,L),
          Cheese = [Choice].
/*choose as many as you want (except for value meal)*/
chooseVeg(Veg,Option):-
          format("Would you like some vegetables?\n"),
          vegList(L),
          options(L),
          healthyOption(Option,L),
          (   value(Option),
              getUserInput(Choice,L),
              Veg = [Choice];
```

```
          getRecursiveInput(Veg,L)).
/*choose as many as you want (again, except for value meal)*/
chooseSauce(Sauce,Option):-
          format("Would you like any sauces?\n"),
          sauceList(L),
          options(L),
          healthyOption(Option,L),
          (   value(Option),
              getUserInput(Choice,L),
              Sauce = [Choice];
          getRecursiveInput(Sauce,L)).
/*choose as many, vegans can't have cookies, value only gets 1*/
chooseSide(Side,Option):-
          format("What sides would you like?\n"),
          (   vegan(Option),
              veganSideList(L);
          sideList(L)),
          options(L),
          healthyOption(Option,L),
          (   value(Option),
              getUserInput(Choice,L),
              Side = [Choice];
           getRecursiveInput(Side,L)).
/*choose 1, vegans can't have milk*/
chooseDrink(Drink,Option):-
          format("Please choose a drink:\n"),
          (   vegan(Option),veganDrinkList(L);
              drinkList(L)),
          options(L),
          healthyOption(Option,L),
          getUserInput(Choice,L),
          Drink = [Choice].
/*display order*/
checkout(Meal,Type,Option):-
          format('Your ~a ~a:\n',[Option,Type]),
          format("---------------\n"),
          (   sandwich(Type),
              printSandwich(Meal,_);
          combo(Type),
              printSandwich(Meal,Other),
              printOther(Other);
          other(Type),

            printOther(Meal)),
          format("---------------\n"),
          format("Thank you for choosing Subway!\n").
/*print sides and drinks unless "none" option*/
printNormal(B):-
          B = "none";
          format('|~a\n',[B]).
/*print toppings with indent unless "none" option*/
printTopping(B):-
          B = "none";
          format('|   ~a\n',[B]).
/*print sandwich type and toppings*/
printSandwich(A,D):-
          A = [B,S|T],
          format('|~a sandwich on ~a\n',[S,B]),
          printSandwichToppings(T,D).
/*recursive function, print all sandwich toppings
  Stops when list is empty or a side/drink appears*/
printSandwichToppings([],[]).
printSandwichToppings(A,D):-
          (   A = [B|C],
              (proteinList(L);cheeseList(L);vegList(L);sauceList(L)),
              member(B,L),
              printTopping(B),
              printSandwichToppings(C,D);
          A = D).
/*print all sides and drinks in the order.
  Stops when the list is empty.*/
printOther([]).
printOther(A):-
          A = [B|C],
          printNormal(B),
          printOther(C).
/*ValidString is a string typed by the user.
  Only returned if it is a member of the list List,
  otherwise it asks again.*/
getUserInput(ValidString,List):-
          read_line_to_string(user_input,String),
          (   member(String,List),ValidString = String;
              format("This is not an option. Please try again:\n"),
              getUserInput(ValidString,List)).

/*Choice is a list of strings ["s1","s2",...]
  It calls recursiveChoice, which asks user for input
  until the user types "none" or is out of options.*/
getRecursiveInput(Choice,L):-
          getUserInput(Input,L),
          (   Input = "none",Choice = ["none"];
          takeout(Input,L,SL),
              recursiveChoice(More,SL),
              append([Input],More,Choice)).
recursiveChoice([],["none"]).
recursiveChoice(Choice,L):-
          format("Would you like any more?\n"),
          options(L),
          getUserInput(Input,L),
          (   Input = "none",Choice = [];
          takeout(Input,L,SL),
              recursiveChoice(More,SL),
              append([Input],More,Choice)).
```