CMPE 260 - Principles of Programming Languages
Spring 2015
Project 1

Due date: March 29, 23:59

1 Personal Meal Planner

When you go to a restaurant, you have to go through many meals in the menu and it is hard to choose which one to pick, especially if you have a special dietary requirement. For this reason, you will design an interface for the user so that given a customer name, the program will list all the meals that the customer would be willing to eat.

2 Knowledge Base

In your knowledge base (plannerData.pl), you have a list of predicates, *foodGroup*, *meal* and *customer*. They are defined as follows:

```
   foodGroup(+GroupName, +FoodList).
   cannotEatGroup(+EatingType, +CannotEatFoodGroupList, +CalorieLimit).
   meal(+MealName, +IngredientList, +Calorie, +PrepTime, +Price).
   customer(+CustomerName, +AllergyList, +EatingType, +Dislikes, +Likes, +TimeInHand,
+MoneyInHand).
```

- **foodGroup** defines a food group, and for each food group, list of foods belonging to that category are listed, e.g.

  `foodGroup(vegetable, [cauliflower, spinach, potato, zucchini, onion, lettuce]).`

  Keep in mind that *egg* is categorized as *meat*, which could be confusing, but it's categorized under the meat and protein group in the food pyramid.

- **cannotEatGroup** defines the properties of the EatingType (normal, vegetarian, vegan, diabetic, diet, etc.), e.g.

  `cannotEatGroup(vegan,[meat, dairy], 0).`

  `cannotEatGroup(diet, [], 220).`

  If *CalorieLimit* is different than 0, then the group cannot eat meals with *Calorie* ¿ *CalorieLimit*, else if *CalorieLimit* is 0, there is no limit to *Calorie* intake. *CannotEatFoodGroupList* consists of the list of *foodGroups* that the *EatingType* cannot eat.

- **meal** defines a meal and it consists of meal name, ingredients list, calorie of the meal in kcal, preparation of the meal in the restaurant (in minutes), and price of the meal in TL, e.g.

  ```
  meal(chickenWrap,[chicken,lettuce,corn,tomato,cucumber,yogurt,flour,salt],180,
  20,15).
  ```

- **customer** holds the customer name and preferences, namely their list of allergies (as foodGroup), eating type (normal, vegetarian, vegan, diabetic, diet), dislikes (as FoodList (**NOT** foodGroup)), likes (as FoodList), the time they have available to wait for the meal to come (in minutes),and the money they have in hand to pay (in TL).

  ```
  customer(nick, [cheese], [diet], [fish, tomato], [chicken, walnut], 15, 20).
  ```

3 Planner

In the planner section, you are required to write some predicates for this project. *InitialList* is used to hold the current state of the meal list (starts from total list and might decrease in size as preferences are satisfied).

3.1 findAllergyMeals(+AllergyList, ?InitialList, -MealList)

This predicate will be used to find the meals that contains a foodGroup that the customer is allergic to. AllergyList can contain more than one allergy, e.g. [fruit], [nut, cheese], etc.

Example:

```
   findAllergyMeals([nut],[muesli,eggSandwich,saladWithNuts,tomatoSoup,bananaCake],
MealList).
   MealList = [muesli, saladWithNuts].
```

3.2 findLikeMeals(+Likes, ?InitialList, -MealList)

This predicate is to find the meals that the customer likes/dislikes. *Likes* is a list of foods, e.g. [tomato, banana, egg, ...].

Example:

```
   findLikeMeals([tomato,banana,egg],[muesli,eggSandwich,saladWithNuts,tomatoSoup,b
ananaCake],MealList).
   MealList = [eggSandwich,saladWithNuts,tomatoSoup,bananaCake].
```

3.3 findNotEatingTypeMeals(+EatingTypeList, ?InitialList, -MealList)

This predicate finds the meals that the customer wouldn't eat. The properties for EatingType is defined by

```
   cannotEatGroup(EatingType, CannotEatFoodGroupList, CalorieLimit).
```

EatingTypeList can be a combination of *EatingTypes*, e.g. [normal], [vegan, diabetic] or [normal, diet, diabetic], etc.

Example:

```
findNotEatingTypeMeals([diet],[muesli,eggSandwich,saladWithNuts,tomatoSoup,banan
aCake,karniyarik,friedZucchini],MealList).
MealList = [friedZucchini, karniyarik].
```

3.4 `findMealsForTime(+TimeInHand, ?InitialList, -MealList)`

This predicate finds the meals that the customer has time to wait to eat, that is, the customer must have more or same amount of time in hand than the preparation time of the meal.

Example:

```
findMealsForTime(15,[saladWithNuts,tomatoSoup,ayran,karniyarik,friedZucchini],Me
alList).
MealList = [saladWithNuts, tomatoSoup, ayran].
```

3.5 `findMealsForMoney(+MoneyInHand, ?InitialList, -MealList)`

This predicate finds the meals that the customer has the money to pay for, that is, the customer must have more or same amount of money in hand than the price of the meal.

Example:

```
findMealsForMoney(15,[saladWithNuts,tomatoSoup,ayran,karniyarik,friedZucchini],M
ealList).
MealList = [tomatoSoup,ayran,karniyarik,friedZucchini].
```

3.6 `orderLikedList(+LikeMeals, ?InitialList, -MealList)`

This predicate puts the meals that the customer likes to the beginning of the list (*LikeMeals* list comes from `findLikeMeals(+Likes, ?InitialList, -MealList)`. Read the NOTE section below for more explanation on ordering.

Example:

```
orderLikedList([karniyarik,tomatoSoup,saladWithNuts],[saladWithNuts,tomatoSoup,a
yran,karniyarik,friedZucchini],MealList).
MealList = [karniyarik, tomatoSoup, saladWithNuts, ayran, friedZucchini].
```

3.7 `listPersonalList(+CustomerName, -PersonalList)`

This is the main predicate with which you will extract the personal meal list for a customer. PersonalList found from all available meals in the knowledge base.

NOTE: The PersonalList must be in the same order as in the knowledge base, except if the customer has specific Likes. If the customer has Likes, than their liked meals should appear in the prioritized order but as in the knowledge base list. E.g. if

`Likes = [chocolate, banana].`

Then

`MealList = [chocolateBrownie, bananaOatmeal, bananaCake, ...]`

`...` stands for rest of the list in the order as in the knowledge base but not including the first three meals. chocolateBrownie came first, because it has more priority to banana in the Likes list (i.e. it comes first), than bananaOatmeal is and bananaCake is given as in the order in the knowledge base list. See Test Cases for the full list.

3.8 Test Cases

NOTE: Your projects will be graded with similar test cases from plannerData.pl file and also with another knowledge base (with different foodGroup, cannotEatGroup, meal and customer predicates) and different values for Food and EatingTypes, so don't encode any meal name, customer name, foodGroup, etc. in your code, otherwise you will get a ZERO! Take everything from the knowledge base (plannerData.pl file) and use variable names (**NOT** values) for your predicates and rules.

1. `listPersonalList(amy, PersonalList).`

   `PersonalList = [muesli,bananaOatmeal,cheeseCrepe,eggSandwich,saladWithNuts,tomatoSoup,friedZucchini,karniyarik,chickenWrap,fishSticks,salmon,bananaCake,hazelnutBrownie,chocolateBrownie,tea,ayran].`

2. `listPersonalList(jack,PersonalList).`

   `PersonalList =[cheeseCrepe,eggSandwich,tomatoSoup,friedZucchini,karniyarik,chickenWrap,fishSticks,salmon,bananaCake,chocolateBrownie,tea,ayran].`

3. `listPersonalList(melanie,PersonalList).`

   `PersonalList = [muesli,bananaOatmeal,saladWithNuts,tomatoSoup,friedZucchini,tea,ayran].`

4. `listPersonalList(wendy,PersonalList).`

   `PersonalList = [muesli,bananaOatmeal,cheeseCrepe,eggSandwich,saladWithNuts,tomatoSoup,chickenWrap,salmon,bananaCake,tea,ayran].`

5. `listPersonalList(john,PersonalList).`

   `PersonalList = [saladWithNuts,friedZucchini,tea].`

6. `listPersonalList(bailey,PersonalList).`

   `PersonalList = [muesli,bananaOatmeal,cheeseCrepe,eggSandwich,tomatoSoup,karniyarik,chickenWrap,fishSticks,salmon,bananaCake,hazelnutBrownie,chocolateBrownie,tea,ayran].`

7. `listPersonalList(sarah,PersonalList).`

   `PersonalList = [chocolateBrownie,bananaOatmeal,bananaCake,muesli,cheeseCrepe,eggSandwich,saladWithNuts,tomatoSoup,friedZucchini,karniyarik,chickenWrap,fishSticks,salmon,hazelnutBrownie,tea,ayran].`

8. `listPersonalList(brad,PersonalList).`

   `PersonalList = [muesli,bananaOatmeal,tomatoSoup,bananaCake,hazelnutBrownie,chocolateBrownie,tea,ayran].`

9. `listPersonalList(alison,PersonalList).`

   `PersonalList = []`

10. `listPersonalList(nick,PersonalList).`

    `PersonalList = [bananaOatmeal,muesli,bananaCake,tea,ayran].`

4 Efficiency

Your code should run under one minute for all the test cases collectively.

5 Documentation

You will be graded for the readability of your code, so don't skip steps or make your code complex, write as clear as possible. You will also be graded for the documentation of your code, so explain what each predicate is for in comments in your code.

6 Submission

You are going to submit *planner.pl* and a plain text file named *status.txt* in a zip file named as 260-Prolog-YOUR_STUDENT_ID.zip to **b.irfann@gmail.com** . The *status.txt* file must have exactly the syntax below, since it will be used for compiling and testing your code automatically:

```
compiling:  yes
complete:  yes
```

The first line denotes whether your code compiles correctly, and the second line denotes whether you completed all of the project, which must be "no" (without quotes) if youre doing a partial submission. The whole file must be lowercase.

NOTE ON LATE SUBMISSIONS: 15 points will be deducted for each day you submit your project late, i.e. 15 points for submitting one day later, 30 points will be deducted for submitting 2 days later, and so on. It's advised that you start your project as early as possible.

7 Tips for the Project

- Try to formalize the problem, specially the predicates that need to find a set, then try to convert the logic formulate to Prolog.
- You can use `findall/3`, `bagof/3` or `setof/3`. Be careful when using `bagof/3` and `setof/3`, and remember to set which free variables to ignore.

- You can use extra predicates for listing all the meals or removing lists, but the ones given above are compulsory.
- You can take a look at PS slides and 99 Prolog Problems.
- Try to build complex predicates over the simpler ones, the project is designed to encourage that.
- If a predicate becomes too complex, either divide it into some predicates, or take another approach.
- Use debugging (through `trace/1` and `spy/1`), approach your program systematically. A nice article about importance of debugging can be found at `http://danluu.com/teach-debugging/`.
- You can use ordered sets (`ordsets`) to use set operations like intersection etc. The documentation on `ordsets` can be found in `http://swi-prolog.org/pldoc/man?section=ordsets`
  - Ordered sets are actually just lists when you look from the outside.
  - `sort/2` and `setof/2` produce ordered sets, so if you need to convert a list to an ordered set you can use sort/2.