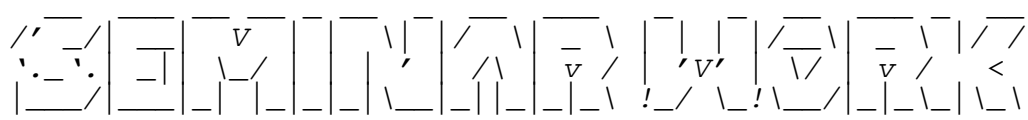
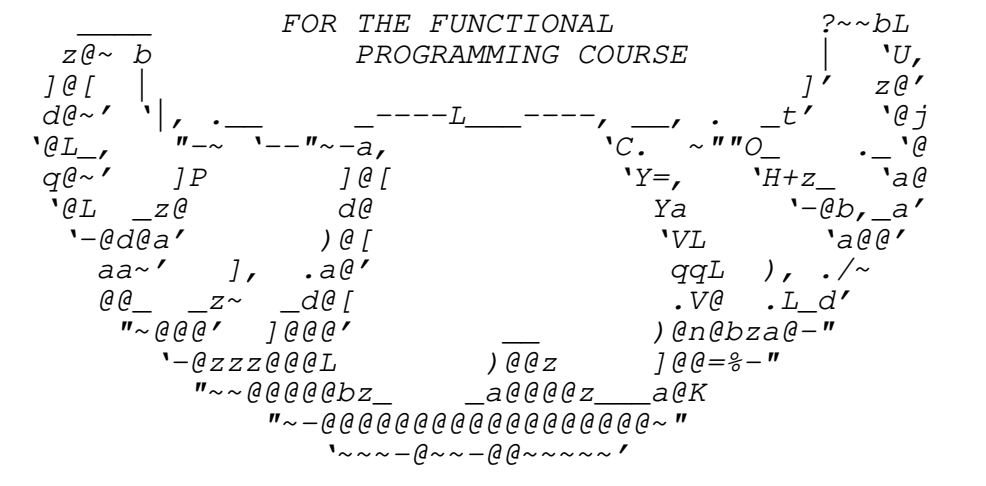
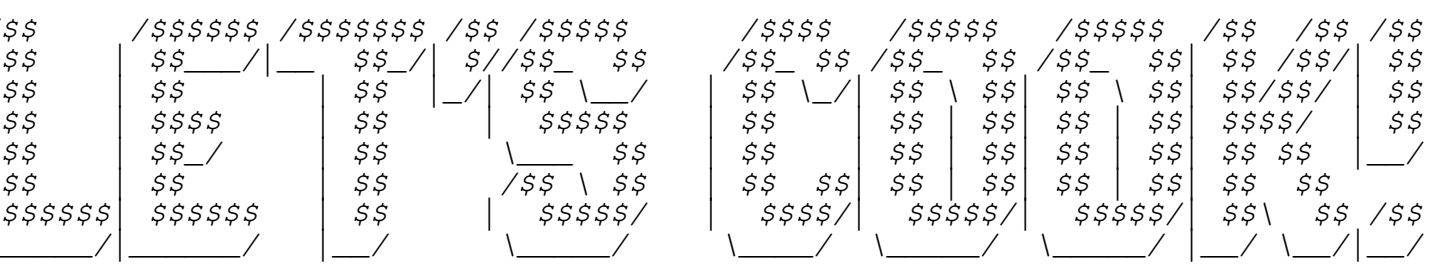
(*







For this seminar work you will write a program to help you with managing ingredients, recipes, prices and cookbooks. The seminar is divided into two parts. In the first part you will implement a dictionary data structure, which you will use in the second part for the implementation of gourmet methods.

The assignment is organized into three files:

- *cookbook-si.sml and cookbook-en.sml* — *These files hold the signatures of the structures you have to implement. DO NOT change these files!*
- *solution.sml* — *In this file you will implement the required structures. The outline is already given to get you started.*
- *tests.sml* — *Write your tests in this file. Try to test your methods thoroughly with different inputs, including edge cases.*

A Makefile is also included in the assignment, and is already set up so that it runs your tests.

Grading:

- *Dictionary: 20 points*
- *Cookbook: 70 points*
- *Defense: 10 points*

Submit your seminar work to e-ucilnica in a file named <student nr.>-sem1.zip, which should hold the files solution.sml and tests.sml in the root directory

```
(without any subdirectories).

*)




(* The signature DICTIONARY defines a type and a programming interface for
   the dictionary data structure. The data structure allows us to store
   data in the form of (key, value) pairs and to query the data using a key. *)
signature DICTIONARY =
sig
    (* The structure has to implement a dictionary type. It defines key type,
       which has to support equality checking, and a value type for the data
       stored in the dictionary. *)
    type (''key, 'value) dict

    (* Creates an empty dictionary. *)
    val empty: (''key, 'value) dict

    (* Returns true if a key exists in the dictionary. *)
    val exists: (''key, 'value) dict → ''key → bool

    (* Returns true if the dictionary is empty. *)
    val isEmpty: (''key, 'value) dict → bool

    (* Returns the number of (key, value) pairs in the dictionary. *)
    val size: (''key, 'value) dict → int

    (* Returns SOME value corresponding to the specified key
       or NONE if the key doesn't exist. *)
    val get: (''key, 'value) dict → ''key → 'value option

    (* Returns the value corresponding to the specified key
       or a default value if the key doesn't exist. *)
    val getOrDefault: (''key, 'value) dict → ''key → 'value → 'value

    (* Stores the (key, value) pair and returns the updated dictionary.
       If a pair with the specified key already exists, then it overwrites
       the existing value. *)
    val set: (''key, 'value) dict → ''key → 'value → (''key, 'value) dict

    (* Removes a (key, value) pair with the specified key and
       returns the updated dictionary. *)
    val remove: (''key, 'value) dict → ''key → (''key, 'value) dict

    (* Returns a list of keys in the dictionary. *)
    val keys: (''key, 'value) dict → ''key list

    (* Returns a list of values in the dictionary. *)
    val values: (''key, 'value) dict → 'value list

    (* Returns a list of (key, value) pairs in the dictionary. *)
    val toList: (''key, 'value) dict → (''key * 'value) list

    (* Creates a dictionary from a list of (key, value) pairs.
       If there are multiple pairs with the same key,
       the rightmost value overrides all previous values. *)
    val fromList: (''key * 'value) list → (''key, 'value) dict

    (* Overrides or adds (key, value) pairs from
       the right dictionary into the left one. *)
```

```
    val merge: (''key, 'value) dict
        → (''key, 'value) dict
        → (''key, 'value) dict

    (* Keeps only the (key, value) pairs for which
       the specified function returns true. *)
    val filter: ((''key * 'value) → bool)
        → (''key, 'value) dict
        → (''key, 'value) dict

    (* Maps the (key, value) pairs with the specified function.
       If the function is not injective with respect to the key,
       the results are undefined. *)
    val map: ((''key * 'value) → (''newkey * 'newvalue))
        → (''key, 'value) dict
        → (''newkey, 'newvalue) dict
end




(* The signature COOKBOOK defines types and a programming interface to work
   with ingredients, stock, recipes, prices and cookbooks. *)
signature COOKBOOK =
sig
    (* An ingredient has a name. *)
    type ingredient

    (* A stock is a mapping from ingredients to their quantities. *)
    type stock

    (* A pricelist holds the price for one unit of each ingredient. *)
    type pricelist

    (* A recipe creates an ingredient and uses some stock. *)
    type recipe

    (* A cookbook holds the required ingredients for each recipe. *)
    type cookbook

    exception NoPriceException

    (* Create an ingredient from a name. *)
    val makeIngredient: string → ingredient

    (* Create a stock from a list of (ingredient, quantity) pairs. *)
    val makeStock: (ingredient * int) list → stock

    (* Create a price list from a list of (ingredient, price) pairs. *)
    val makePricelist: (ingredient * real) list → pricelist

    (* Create a recipe from a name and a stock of ingredients. *)
    val makeRecipe: (ingredient * stock) → recipe

    (* Create a cookbook from a list of recipes. *)
    val makeCookbook: recipe list → cookbook

    (* Print the ingredient to a string. *)
    val ingredientToString: ingredient → string

    (* Print the stock to a string in the following form:
```

```
        firstIngredientName: firstIngredientQuantity
        secondIngredientName: secondIngredientQuantity
        ...: ...

    Ingredients have to be sorted alphabetically (ASCII order).
    Don't print lines with negative or zero values.
    Don't forget newlines. *)
val stockToString: stock → string

(* Print the pricelist to a string in the following form:

        firstIngredientName: firstIngredientPrice
        secondIngredientName: secondIngredientPrice
        ...: ...

    Ingredients have to be sorted alphabetically (ASCII order).
    Don't forget newlines. *)
val pricelistToString: pricelist → string

(* Print the recipe to a string in the following form:

        === recipeName ===
        firstIngredientName: firstIngredientQuantity
        secondIngredientName: secondIngredientQuantity
        ...: ...

    Ingredients have to be sorted alphabetically (ASCII order).
    Don't forget newlines. *)
val recipeToString: recipe → string

(* Print the cookbook to a string in the following form:

        === firstRecipeName ===
        firstIngredientName: firstIngredientQuantity
        secondIngredientName: secondIngredientQuantity
        ...: ...

        === secondRecipeName ===
        firstIngredientName: firstIngredientQuantity
        secondIngredientName: secondIngredientQuantity
        ...: ...

    Recipes have to be sorderd alphabetically (ASCII order).
    Ingredients in each recipe have to be sorted
    alphabetically (ASCII order).
    Don't forget newlines. *)
val cookbookToString: cookbook → string

(* Returns true if there's enough ingredients in the stock
    for the specified recipe. *)
val hasEnoughIngredients: stock → recipe → bool

(* Cooks the ingredients in the recipe and returns
    an updated stock with the product of the recipe added
    and its ingredients removed. *)
val cook: recipe → stock → stock

(* Returns the price of a stock - the sum of the prices of
    the ingredients according to the specified pricelist. If
    an ingredient is not on the price list, the function should
    raise NoPriceException. *)
val priceOfStock: stock → pricelist → real
```

```
    (* Returns the price of a recipe - the sum of the prices of
       the ingredients according to the specified pricelist. If
       an ingredient is not on the price list, the function should
       raise NoPriceException. *)
    val priceOfRecipe: recipe → pricelist → real

    (* Returns the number of ingredients that are missing for
       a given recipe. If there's enough ingredients available
       in the stock, it returns an empty stock. *)
    val missingIngredients: recipe → stock → stock

    (* Returns a list of products from the cookbook that we can
       cook with a given stock - the recipes that have no missing
       ingredients. Each recipe in the cookbook should be examined
       independently, not sequentially. In other words, the recipes
       we examined before have no effect on the outcome for the
       current recipe. *)
    val possibleRecipes: cookbook → stock → cookbook

    (* Returns a cookbook of all recipes we can cook given a set of
       possible substitutions. The substitutions are given in the form
       of equivalence classes:

       [
           [chicken, beef, veal, pork],
           [red pepper, green pepper],
           ...
       ]

       If the recipe requires e.g. a chicken, we can use pork instead.
       If the recipe requires e.g. a green pepper, we can use
       a red pepper instead. There can be multiple substitutions
       (e.g. pepper and meat).

       - The returned cookbook must also contain the recipe without
         any substitutions.
       - The recipe may contain ingredients that are not present in any
         given equivalence class.
       - You can assume that in an equivalence class each ingredient is
         specified only once.
       - You can assume that a recipe will contain at most one
         ingredient from each equivalence class. *)
    val generateVariants: recipe → ingredient list list → cookbook

    (* Returns the cheapest recipe in the cookbook, or NONE, if the
       cookbook is empty. *)
    val cheapestRecipe: cookbook → pricelist → recipe option
end
```