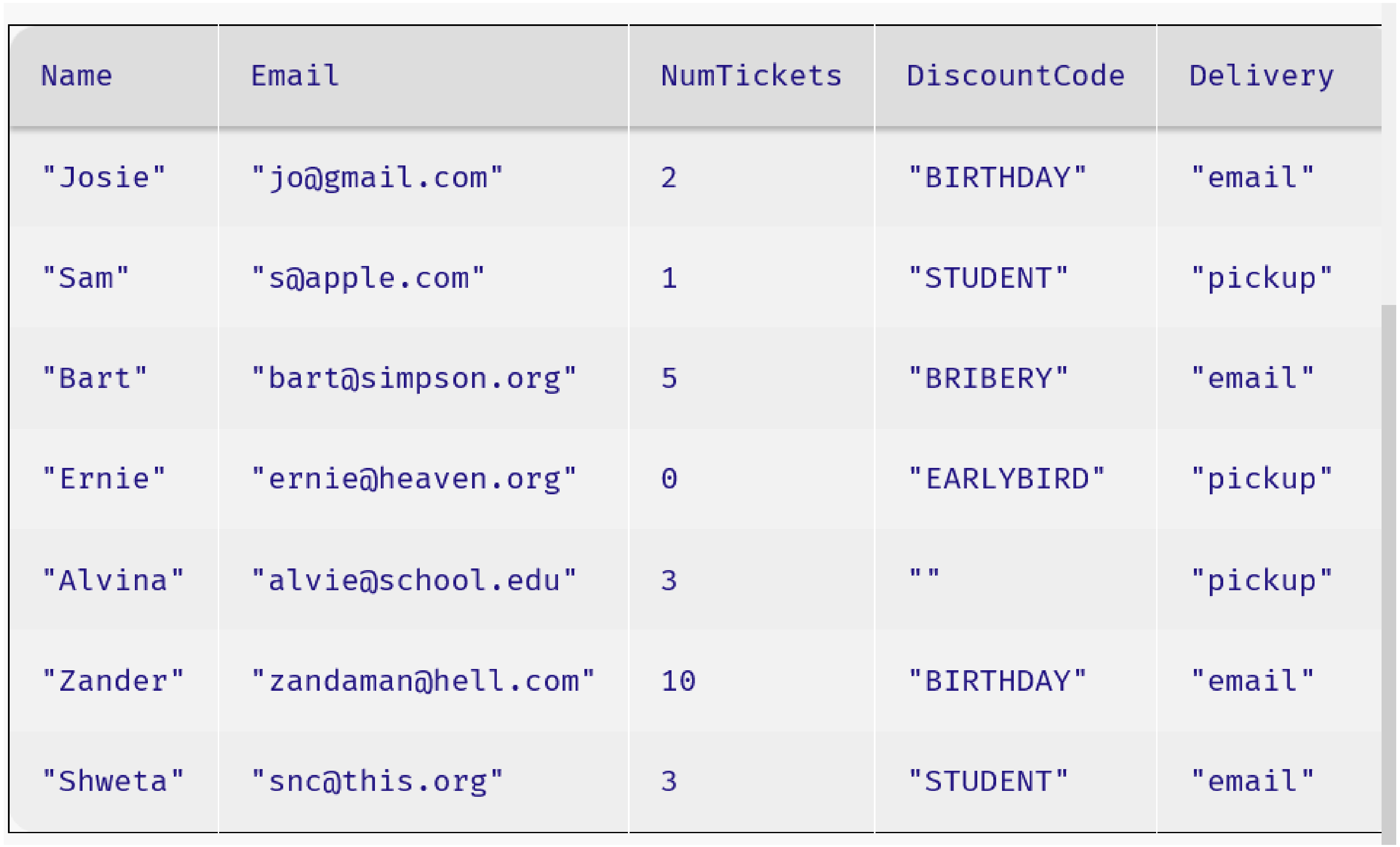
Problem Solving and Abstraction (CMPU 101)

Tom Ellman

Lecture 10

Table Troubles

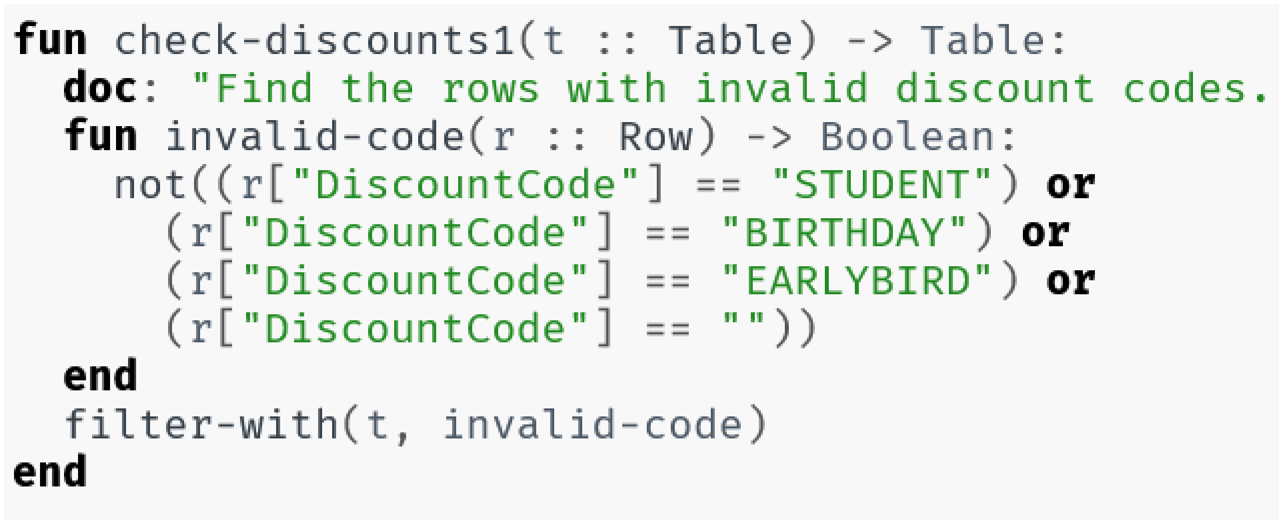


Name	Email	NumTickets	DiscountCode	Delivery
"Josie"	"jo@gmail.com"	2	"BIRTHDAY"	"email"
"Sam"	"s@apple.com"	1	"STUDENT"	"pickup"
"Bart"	"bart@simpson.org"	5	"BRIBERY"	"email"
"Ernie"	"ernie@heaven.org"	0	"EARLYBIRD"	"pickup"
"Alvina"	"alvie@school.edu"	3	""	"pickup"
"Zander"	"zandaman@hell.com"	10	"BIRTHDAY"	"email"
"Shweta"	"snc@this.org"	3	"STUDENT"	"email"

Does every discount in the table appear in the set of valid discount codes?

Does every discount in the table appear in the set of valid discount codes?

At the moment, we might write:



```
fun check-discounts1(t :: Table) -> Table:
  doc: "Find the rows with invalid discount codes.
  fun invalid-code(r :: Row) -> Boolean:
    not((r["DiscountCode"] == "STUDENT") or
      (r["DiscountCode"] == "BIRTHDAY") or
      (r["DiscountCode"] == "EARLYBIRD") or
      (r["DiscountCode"] == ""))
  end
  filter-with(t, invalid-code)
end
```

Cumbersome!

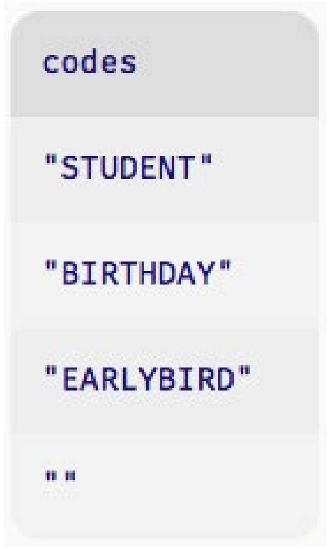
```
fun check-discounts1(t :: Table) -> Table:
  doc: "Find the rows with invalid discount codes.
  fun invalid-code(r :: Row) -> Boolean:
    not((r["DiscountCode"] == "STUDENT") or
      (r["DiscountCode"] == "BIRTHDAY") or
      (r["DiscountCode"] == "EARLYBIRD") or
      (r["DiscountCode"] == ""))
  end
  filter-with(t, invalid-code)
end
```

Every time the set of discount codes changes, we need to change our function.

But how you check the discount-codes shouldn’t change. So let’s write a function that need not change when the data changes.

How can we rewrite this function so the set of valid discount codes is specified outside the function?

Make a table with one column to hold the valid codes?



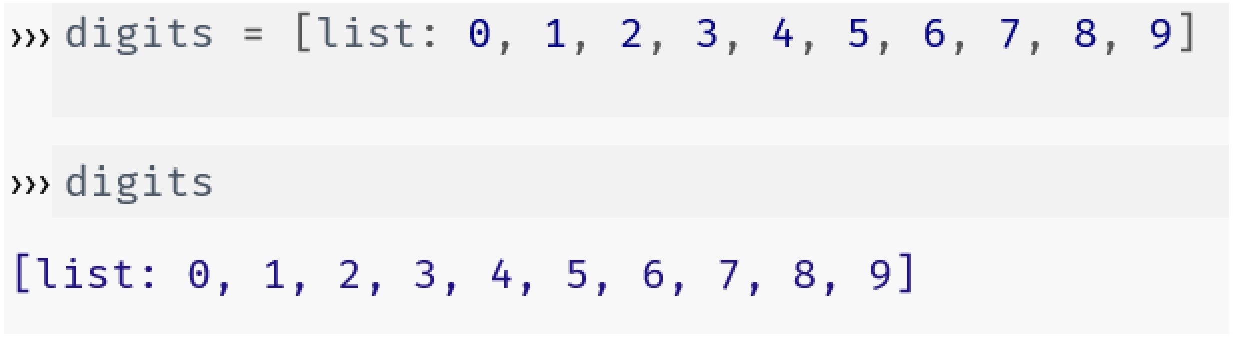
codes
"STUDENT"
"BIRTHDAY"
"EARLYBIRD"
""

This would work, but we really don’t need a table if each row has only one datum.

Lists

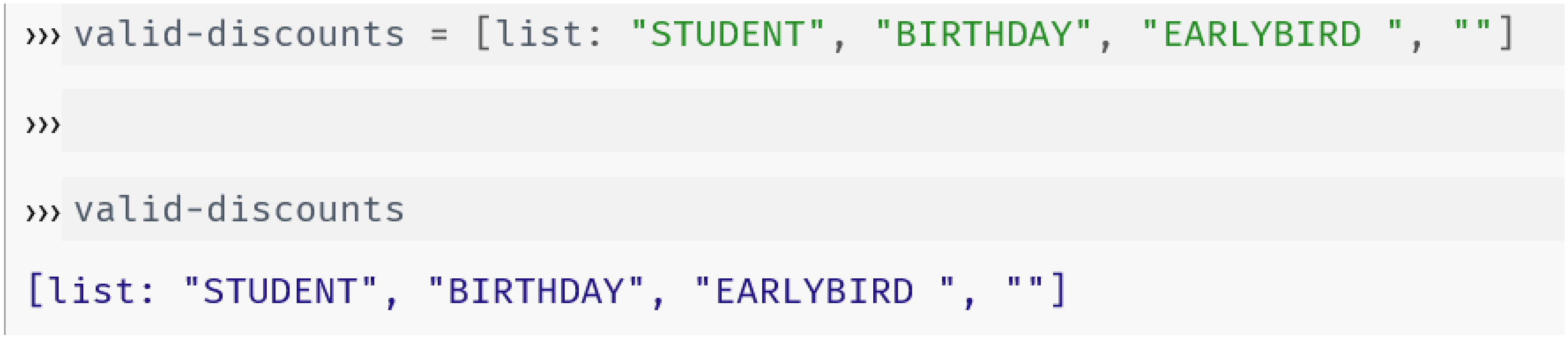
- Lists are a fundamental type of data structure.
- A list is a container type, i.e. a list contains data.
- A datum on a list is called a “member” or an “element” of the list.
- A list can hold any number of elements.
- A list holds elements in a specific order.
- Normally all elements of a list have the same data type.

A list is like a column, but without the header.



```
››› digits = [list: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

››› digits
[list: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



```
››› valid-discounts = [list: "STUDENT", "BIRTHDAY", "EARLYBIRD ", ""]
›››
››› valid-discounts
[list: "STUDENT", "BIRTHDAY", "EARLYBIRD ", ""]
```

```
import lists as L
```

To work with lists, we import the **lists** library and we give it a special name – **L** – to avoid conflicts between the names of functions that work with lists and existing functions.

To use a function from the library, we pre-pend the function name with "**L.**".

Tools for Working with Lists

? -> List

- [list: ...]
- get-column

List -> List

- L.distinct
- L.filter
- L.append

List -> ?

- L.length
- L.member
- L.any
- L.all

Does every discount in the table appear in the set of valid discount codes?

Does every discount in the table appear in the set of valid discount codes?

```
valid-discounts = [list: "STUDENT","BIRTHDAY","EARLYBIRD"]

fun check-discounts2(t :: Table) -> Table:
  doc: "Filter out rows whose discount code is not valid."

  fun valid-code(r :: Row) -> Boolean:
    L.member(valid-discounts, r["discount"])
  end

  filter-with(t, valid-code)
end
```

If the valid discounts change, we need only to update the list: valid-discounts. The function code stays the same.

Table Column to List

When we’ve been working with tables we’ve been using the data type Row, but we never saw a Column data type!

Why not? Well, a column consists of an ordered collection of values, of unbounded length.

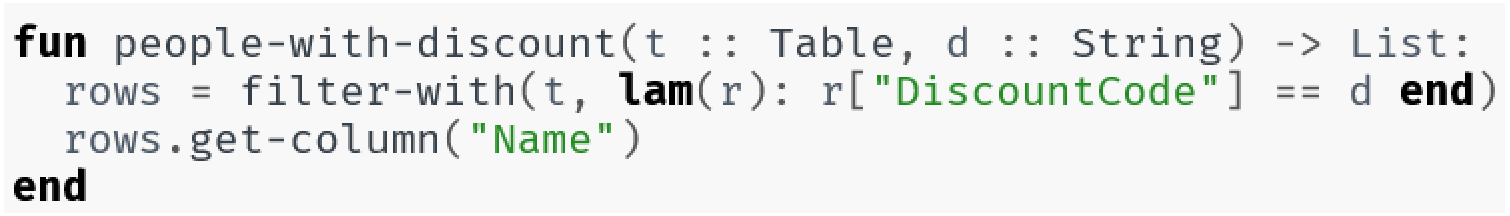
A column is a lot like a list!

```
>>> event-data.get-column("name")
[list: "Josie", "Sam", "Bart", "Ernie", "Alvina", "Zander", "Shweta"]
```

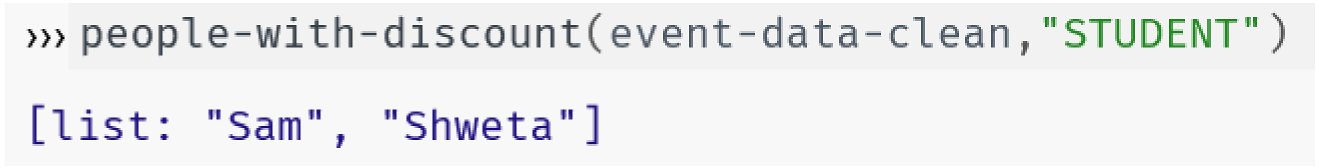
The **get-column** function returns a list.

Find the names of people who got a specific discount type.

Find the names of people who got a specific discount type.

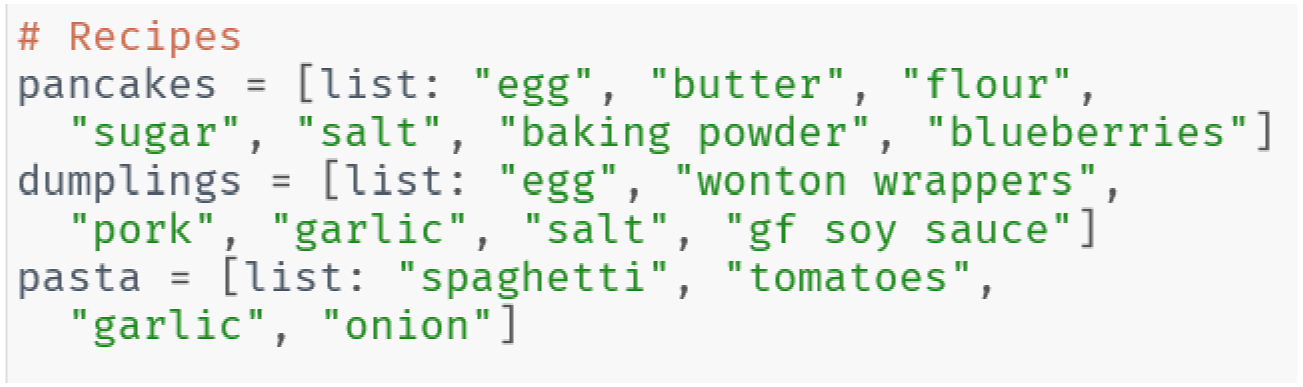


```
fun people-with-discount(t :: Table, d :: String) -> List:
  rows = filter-with(t, lam(r): r["DiscountCode"] == d end)
  rows.get-column("Name")
end
```



```
››› people-with-discount(event-data-clean,"STUDENT")

[list: "Sam", "Shweta"]
```



```
# Recipes
pancakes = [list: "egg", "butter", "flour",
  "sugar", "salt", "baking powder", "blueberries"]
dumplings = [list: "egg", "wonton wrappers",
  "pork", "garlic", "salt", "gf soy sauce"]
pasta = [list: "spaghetti", "tomatoes",
  "garlic", "onion"]
```

A recipe is a list of required ingredients.

```
# Dietary restrictions
gluten = [list: "flour", "spaghetti"]
meat = [list: "chicken", "pork", "beef", "fish"]
dairy = [list: "milk", "butter", "whey"]
eggs = [list: "eggs", "egg noodles"]
```

A dietary restriction is a list of restricted ingredients.

```
# Inventory
pantry = [list: "spaghetti", "wonton wrappers", "garlic"]
```

An inventory is a list of ingredients on hand.

```
meal-ingredients = L.append(pancakes, L.append(dumplings, pasta))
```

All ingredients needed for a (high-carb) meal, possibly with duplicates.

A shopping list is a list of distinct (**L.distinct**) ingredients that are not members (**L.member**) of the inventory list.

```
meal-ingredients = L.append(pancakes, L.append(dumplings, pasta))
```

All ingredients needed for a (high-carb) meal, possibly with duplicates.

```
fun shopping-list(ingredients :: List, inventory :: List)
  -> List:
  d = L.distinct(ingredients)
  L.filter(lam(i): not(L.member(inventory, i)) end, d)
end
```

A shopping list is a list of distinct (**L.distinct**) ingredients that are not members (**L.member**) of the inventory list.

Check whether a recipe is gluten-free using L.filter and L.member.

```
fun is-gluten-free1(recipe :: List<String>) -> Boolean:
  doc: "Return true no ingredients in a list contain gluten."
  non-gf = L.filter(lam(i): L.member(gluten, i) end, recipe)
  L.length(non-gf) == 0
where:
  is-gluten-free1(pancakes) is false
  is-gluten-free1(dumplings) is true
end
```

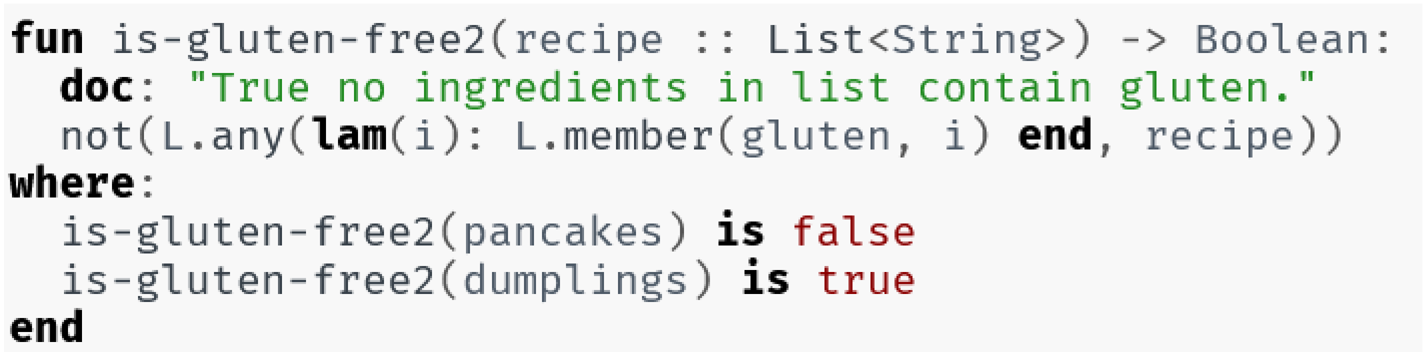
L.filter(<predicate>,<input-list>) returns a new list containing just the members of the input list for which the predicate returns **true**.

In this function we use **L.filter** to make a list of all ingredients in the recipe that are on the gluten list. If the new list is empty (**L.length** is zero), the recipe is gluten-free (return **true**).

Alternate solution using: **L.any**

L.any(<predicate>,<list>) returns **true** if the predicate returns **true** when applied to at least one member of the list.

Write a function to check whether a recipe is gluten-free, using **L.any**.



```
fun is-gluten-free2(recipe :: List<String>) -> Boolean:
  doc: "True no ingredients in list contain gluten."
  not(L.any(lam(i): L.member(gluten, i) end, recipe))
where:
  is-gluten-free2(pancakes) is false
  is-gluten-free2(dumplings) is true
end
```

L.any(<predicate>,<list>) returns **true** if the predicate returns **true** when applied to at least one member of the list.

In this function, we use **L.any** to ensure that there are not any members of recipe that are also members of gluten.

Alternate solution using: **L.all**

L.all(<predicate>,<list>) is **true** if the predicate returns **true** for each member of the list.

Write a function to check whether a recipe is gluten-free, using **L.all**.

```
fun is-gluten-free3(recipe :: List<String>) -> Boolean:
  doc: "True if all ingredients in list don't contain gluten."
  L.all(lam(i): not(L.member(gluten, i)) end, recipe)
where:
  is-gluten-free3(pancakes) is false
  is-gluten-free3(dumplings) is true
end
```

L.all(<predicate>,<list>) is **true** if the predicate returns **true** for each member of the list.

In this function, we use **L.all** to ensure that each member of recipe is not a member of gluten.

Write functions to check whether a recipe is vegan using L.any and with L.all.

```
fun is-vegan1(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the ingredients are non-vegan"
  not(
    L.any(
      lam(i):
        L.member(meat, i) or
        L.member(dairy, i) or
        L.member(eggs, i)
      end,
      recipe))
where:
  is-vegan1(pasta) is true
  is-vegan1(dumplings) is false
end
```

```
fun is-vegan2(recipe :: List<String>) -> Boolean:
  doc: "Return true if all the ingredients are vegan"
    L.all(
      lam(i):
        not(L.member(meat, i) or
          L.member(dairy, i) or
          L.member(eggs, i))
      end,
      recipe)
where:
  is-vegan2(pasta) is true
  is-vegan2(dumplings) is false
end
```

Veganize a Meal

```
pancakes =
        [list: "egg", "butter", "flour",   "sugar",
                "salt", "baking powder", "blueberries"]

vegan-pancakes =
        [list: "flax", "margarine", "flour",
         "sugar", "salt", "baking powder", "blueberries"]
```

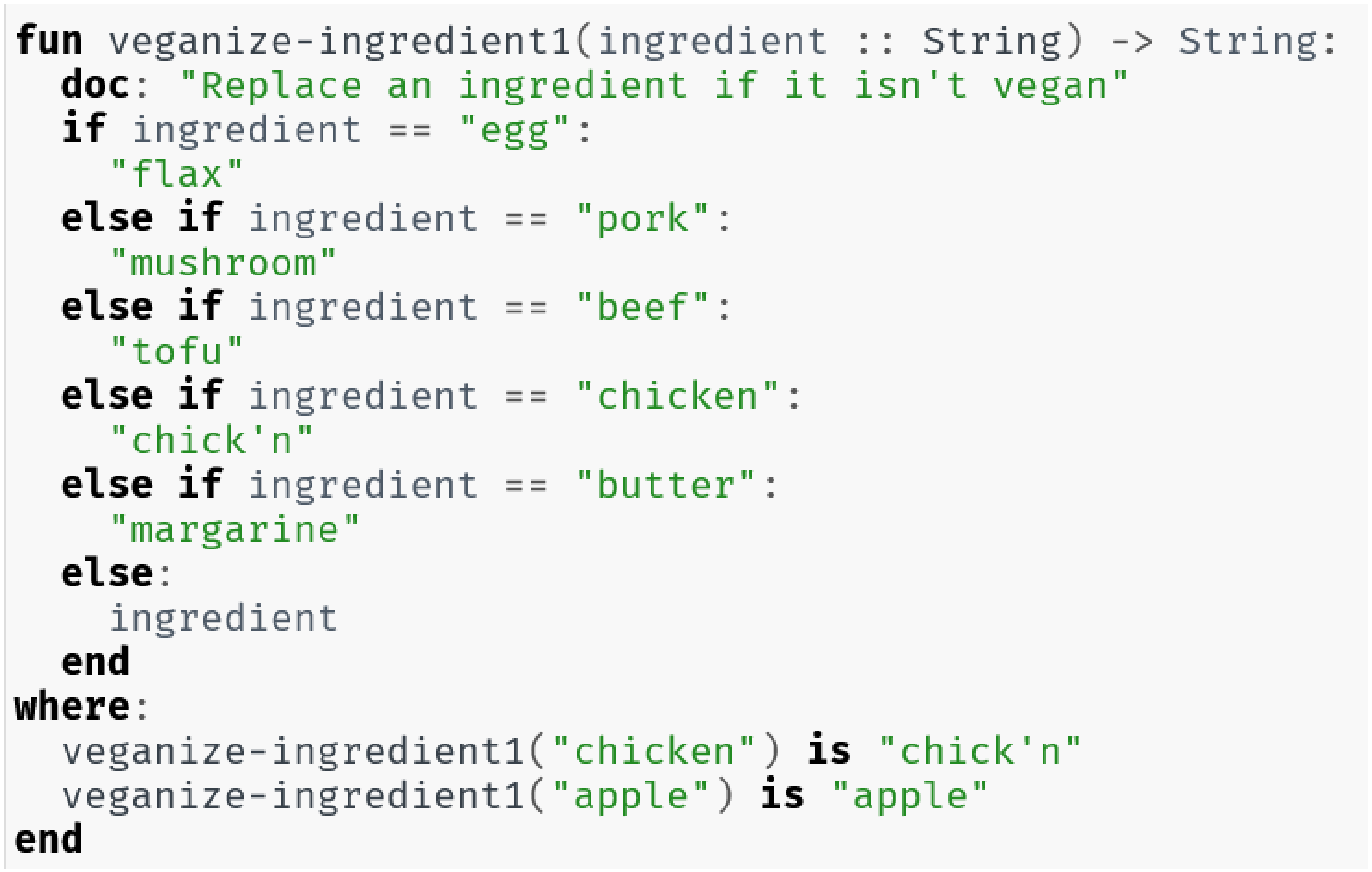
Veganize a Meal

1. Write a function **veganize-ingredient** that takes a non-vegan ingredient and returns something vegan to replace it.

2. Use **veganize-ingredient** repeatedly to replace each non-vegan ingredient on the recipe list. But how can we do this for a recipe list of any length?

Veganize a Meal (Step 1)

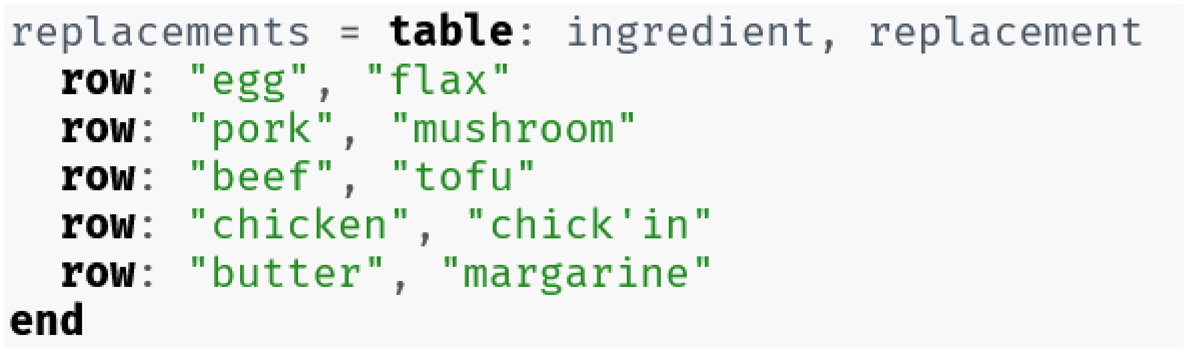
Write a function **veganize-ingredient** that takes a non-vegan ingredient and returns something vegan to replace it.



```
fun veganize-ingredient1(ingredient :: String) -> String:
  doc: "Replace an ingredient if it isn't vegan"
  if ingredient == "egg":
    "flax"
  else if ingredient == "pork":
    "mushroom"
  else if ingredient == "beef":
    "tofu"
  else if ingredient == "chicken":
    "chick'n"
  else if ingredient == "butter":
    "margarine"
  else:
    ingredient
  end
where:
  veganize-ingredient1("chicken") is "chick'n"
  veganize-ingredient1("apple") is "apple"
end
```

Cumbersome! Not easy to update to handle other non-vegan ingredients.

Let’s separate the data from the code, using a table.

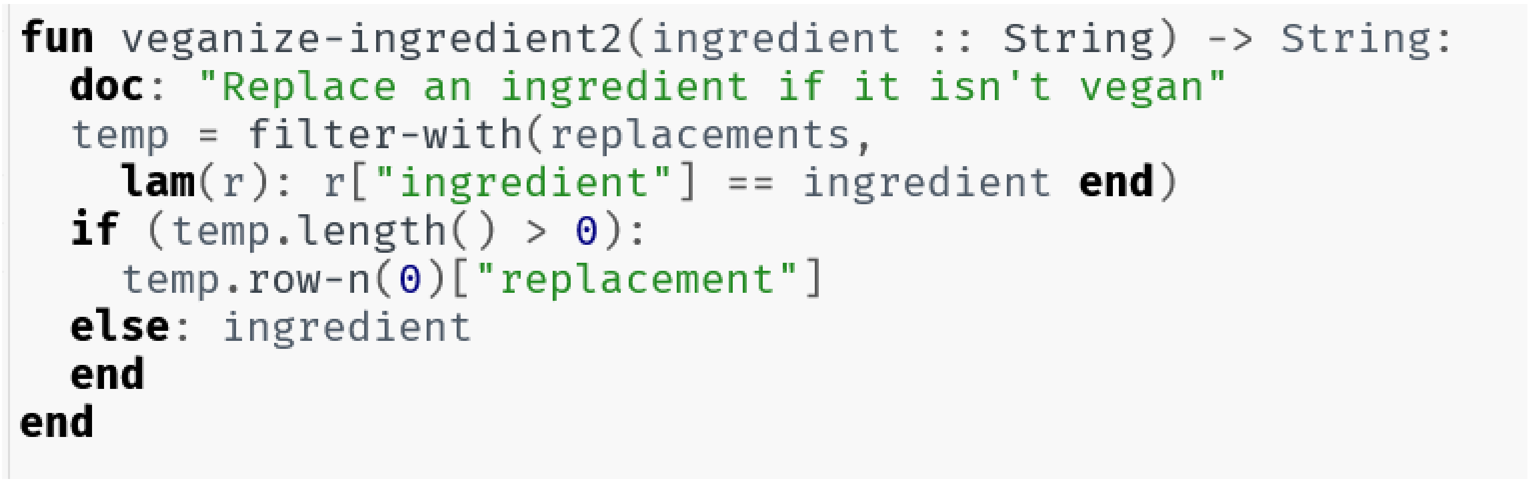


```
replacements = table: ingredient, replacement
  row: "egg", "flax"
  row: "pork", "mushroom"
  row: "beef", "tofu"
  row: "chicken", "chick'in"
  row: "butter", "margarine"
end
```

Each row indicates how a non-vegan ingredient should be replaced by a vegan ingredient.

Could we use a list, or two lists instead of a table?

Rewrite veganize-ingredient1 to use the replacements table.



```
fun veganize-ingredient2(ingredient :: String) -> String:
  doc: "Replace an ingredient if it isn't vegan"
  temp = filter-with(replacements,
    lam(r): r["ingredient"] == ingredient end)
  if (temp.length() > 0):
    temp.row-n(0)["replacement"]
  else: ingredient
  end
end
```

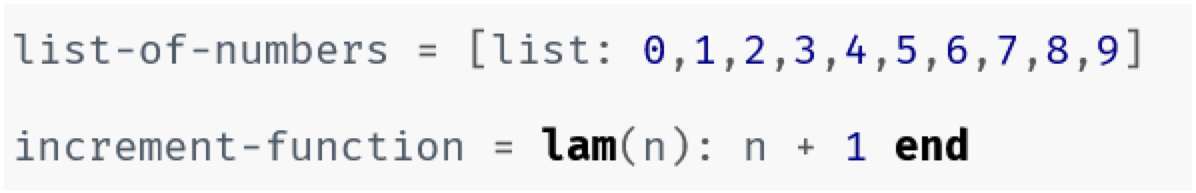
Here we find the replacement by locating a row whose ingredient column matches the ingredient parameter.

```
fun veganize-ingredient2(ingredient :: String) -> String:
  doc: "Replace an ingredient if it isn't vegan"
  temp = filter-with(replacements,
    lam(r): r["ingredient"] == ingredient end)
  if (temp.length() > 0):
    temp.row-n(0)["replacement"]
  else: ingredient
  end
end
```

Here we find the replacement by locating a row whose ingredient column matches the ingredient parameter.

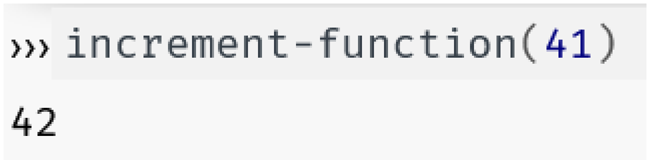
Veganize a Meal (Step 2)

Use **veganize-ingredient** repeatedly to replace each non-vegan ingredient on the recipe list. But how can we do this for a recipe list of any length?

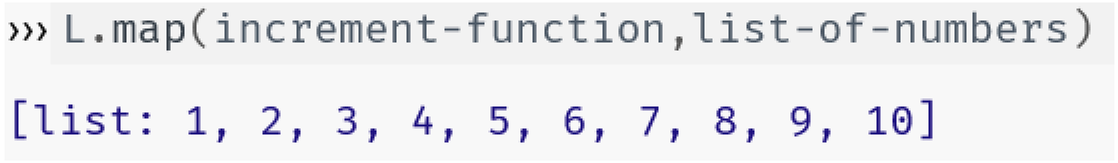


```
list-of-numbers = [list: 0,1,2,3,4,5,6,7,8,9]

increment-function = lam(n): n + 1 end
```



```
››› increment-function(41)
42
```



```
››› L.map(increment-function,list-of-numbers)
[list: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

L.Map takes a function and a list as parameters. It applies the function to each list element and replaces the element with the function result.

Use **L.map** and **veganize-ingredient2** repeatedly to replace each non-vegan ingredient on the recipe list.

```
pancakes =
  [list: "egg", "butter", "flour",   "sugar",
    "salt", "baking powder", "blueberries"]
```

```
replacements = table: ingredient, replacement
  row: "egg", "flax"
  row: "pork", "mushroom"
  row: "beef", "tofu"
  row: "chicken", "chick'in"
  row: "butter", "margarine"
end
```

```
fun veganize-recipe(ingredients :: List<String>) -> List<String>:
  map(veganize-ingredient2, ingredients)
where:
  veganize-recipe(pancakes) is
  [list: "flax", "margarine", "flour",
    "sugar", "salt", "baking powder", "blueberries"]
end
```

L.Map takes a function and a list as parameters. It applies the function to each list element and replaces the element with the function result.

Acknowledgments

This class incorporates material from:

- Marc Smith, Vassar College
- Jason Waterman, Vassar College
- Jonathan Gordon, Vassar College
- Kathi Fisler, Brown University
- Doug Woos, Brown University