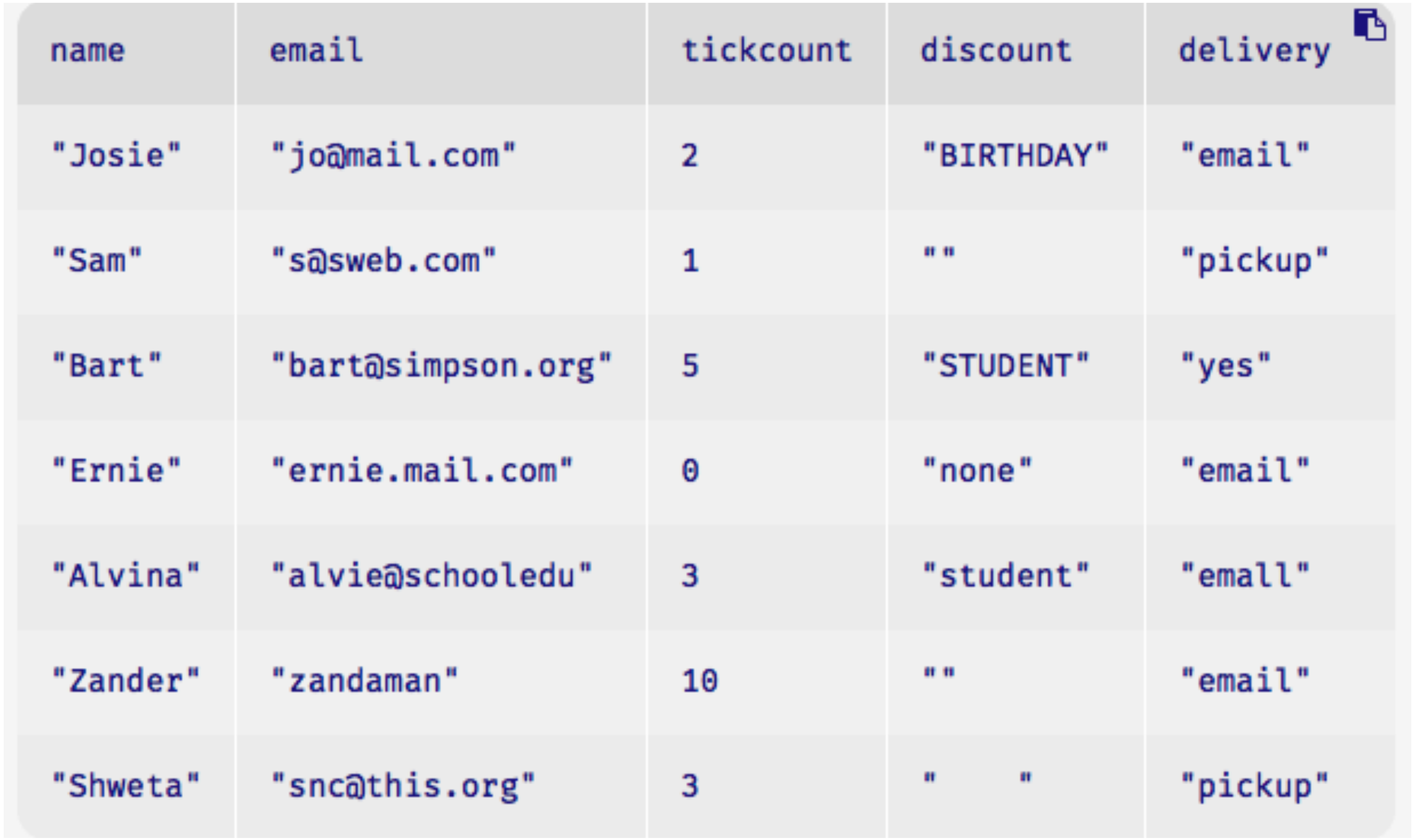
CMPU 101 § 1 · Problem-Solving and Abstraction

Introduction to Lists

4 October 2021

Table trouble



name	email	tickcount	discount	delivery
"Josie"	"jo@mail.com"	2	"BIRTHDAY"	"email"
"Sam"	"s@sweb.com"	1	""	"pickup"
"Bart"	"bart@simpson.org"	5	"STUDENT"	"yes"
"Ernie"	"ernie.mail.com"	0	"none"	"email"
"Alvina"	"alvie@schooledu"	3	"student"	"emall"
"Zander"	"zandaman"	10	""	"email"
"Shweta"	"snc@this.org"	3	" "	"pickup"

Is every discount in the table from a valid set of discount codes?

At the moment, we might write

```
fun check-discounts1(t :: Table) -> Table:
  doc: "Filter out rows whose discount code is not valid"

  fun invalid-code(r :: Row) -> Boolean:
    not(
      (r["discount"] == "STUDENT") or
      (r["discount"] == "BIRTHDAY") or
      (r["discount"] == "EARLYBIRD") or
      (r["discount"] == ""))
  end

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

(plus appropriate test cases!)

At the moment, we might write

```
fun check-discounts1(t :: Table) -> Table:
  doc: "Filter out rows whose discount code is not valid"

  fun invalid-code(r :: Row) -> Boolean:
    not(
      (r["discount"] == "STUDENT") or
      (r["discount"] == "BIRTHDAY") or
      (r["discount"] == "EARLYBIRD") or
      (r["discount"] == ""))
  end

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

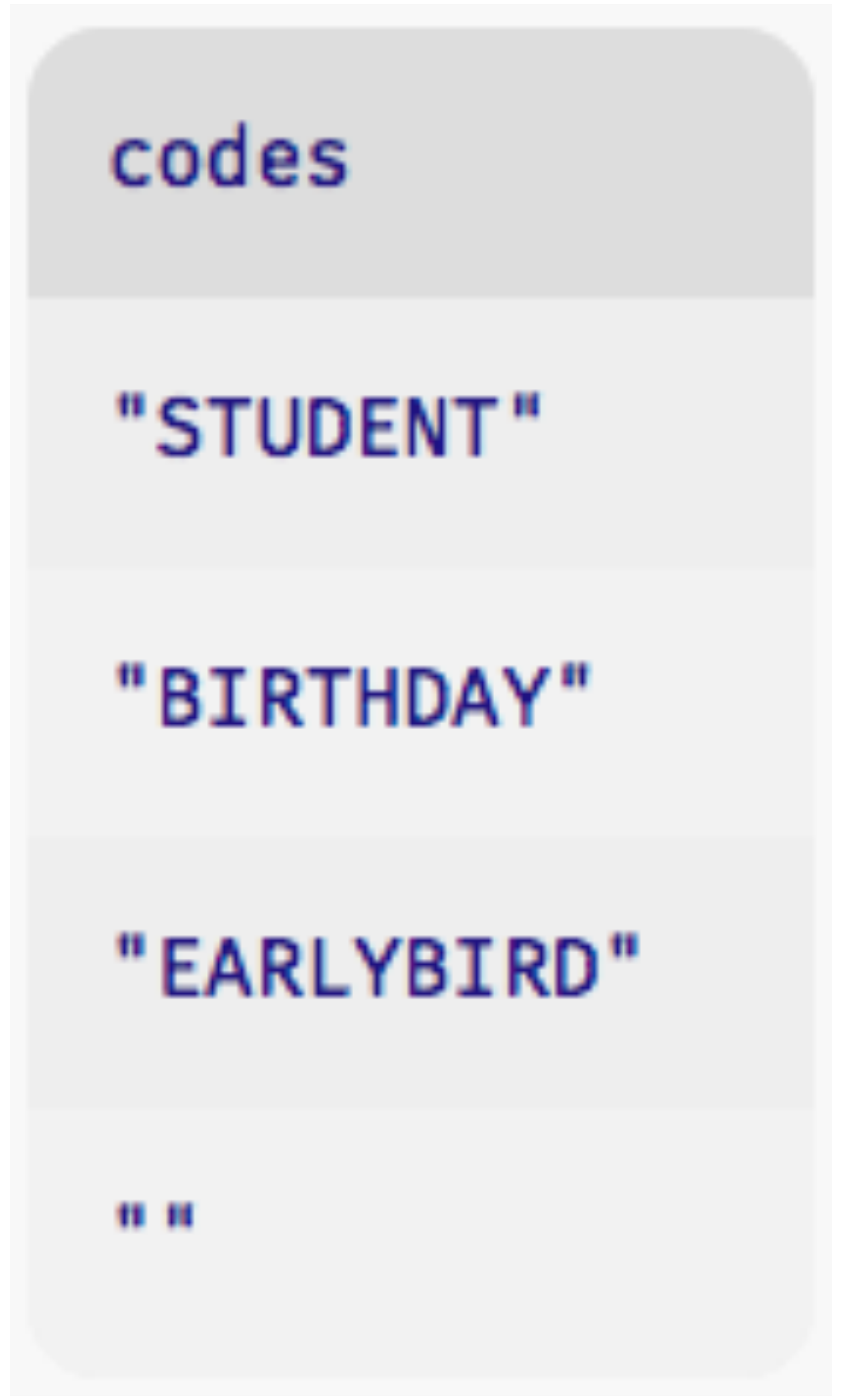
☹️

(plus appropriate test cases!)

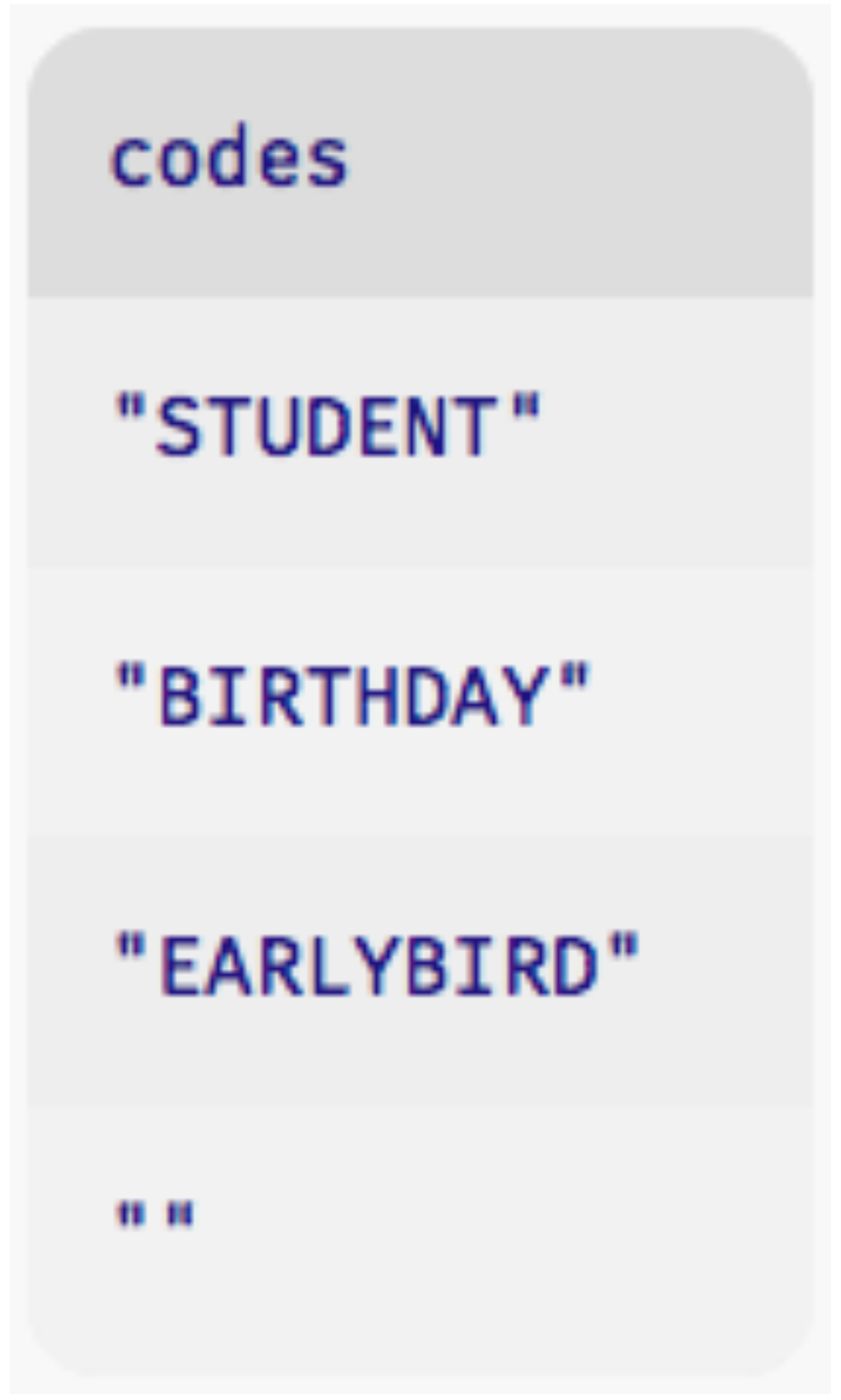
Every time the set of discount codes changes, we need to change our function.

But *how* you check the codes shouldn’t change; it’s just the *data* that’s changing.

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



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



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

Lists to the rescue

Lists are one of the key data structures in programming.

Lists feature:

- An unbounded number of items
- An order on items (first, second, third, …)

A list is like a column of a table, but without the header:

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

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

```
import lists as L
```

We can rewrite our function to check if the discount code in a particular row is one of the valid discount codes, using the **`L.member`** function to check if something is a member of a given list:

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

  fun invalid-code(r :: Row) -> Boolean:
    not(L.member(valid-discounts, r["discount"]))
  end

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

Tables and lists

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 really just a list!

To get a list of values from a column in a table, we can use the **`get-column`** table operator:

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

What if we want the names of everyone who used the `"STUDENT"` discount code?

```
rows =
  filter-with(
    event-data-clean,
    lam(r): r["discount"] == "STUDENT" end)
rows.get-column("name")
```

List operations

You could use lists to keep track of the ingredients used for different 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"]
```

And it would be helpful to know what ingredients we already have:

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

Let’s say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

Let's say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

append *combines two lists, adding one onto the end of the other.*

Let’s say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

shopping-list = L.filter(
  lam(i): not(L.member(pantry, i)) end,
  meal-plan)
```

Let’s say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

shopping-list = L.filter(
  lam(i): not(L.member(pantry, i)) end,
  meal-plan)
```

`filter` is like the `filter-with` function we used on tables: It keeps list members on which its function argument returns true

Let’s say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

shopping-list = L.filter(
  lam(i): not(L.member(pantry, i)) end,
  meal-plan)
```

`member` tells us if the second argument is an item in the specified list.

```
››› shopping-list
[list: "egg", "butter", "flour", "sugar", "salt",
  "baking powder", "blueberries", "egg", "pork",
  "salt", "gf soy sauce", "tomatoes", "onion"]
```

```
››› shopping-list
[list: "egg", "butter", "flour", "sugar", "salt",
  "baking powder", "blueberries", "egg", "pork",
  "salt", "gf soy sauce", "tomatoes", "onion"]
```

```
››› shopping-list
[list: "egg", "butter", "flour", "sugar", "salt",
  "baking powder", "blueberries", "egg", "pork",
  "salt", "gf soy sauce", "tomatoes", "onion"]
```

```
>>> shopping-list
[list: "egg", "butter", "flour", "sugar", "salt",
  "baking powder", "blueberries", "egg", "pork",
  "salt", "gf soy sauce", "tomatoes", "onion"]
```

Let's say we want to go shopping for the ingredients we need to make all three dishes. How would we make such a list?

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

shopping-list = L.filter(
  lam(i): not(L.member(pantry, i)) end,
  L.distinct(meal-plan))
```

distinct gives us a list without the duplicate elements

What if we want to write a predicate that looks at a recipe and returns true if it’s gluten-free?

We can add new lists for ingredients containing gluten – and other dietary concerns:

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

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the 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-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the 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-free(pancakes) is false
  is-gluten-free(dumplings) is tru
end
```

This is an interesting new type annotation!

The input is a List, but we know that each item it contains is a String. If we're given a list of numbers we'll have a problem!

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the 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-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the 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-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

How many elements are in the given list?

Higher-order functions like **`L.filter`** – i.e., functions that take functions as input – are meant to save us effort.

They capture the similarities among many specific functions we *could* write, so we only need to specify the way those functions would differ.

`filter-with` captured the pattern of wanting to filter a table to just the rows that pass some test.

`L.filter` captures the same pattern for lists.

But what we just saw is another common pattern – we want to know whether *any* element passes a test!

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the ingredients in a
list contain gluten"
  not(L.any(lam(i): L.member(gluten, i) end, recipe))
where:
  is-gluten-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the ingredients in a
list contain gluten"
  not(L.any(lam(i): L.member(gluten, i) end, recipe))
where:
  is-gluten-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

any returns true if its function argument returns true on any element of the given list.

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

Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University

Doug Woos, Brown University