

CMPU 101 § 04/05 · Problem-Solving and Abstraction

Introduction to Lists

6 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"	"email"
"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!)

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"

" "



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, ...)

- Many built-in operations on lists

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:

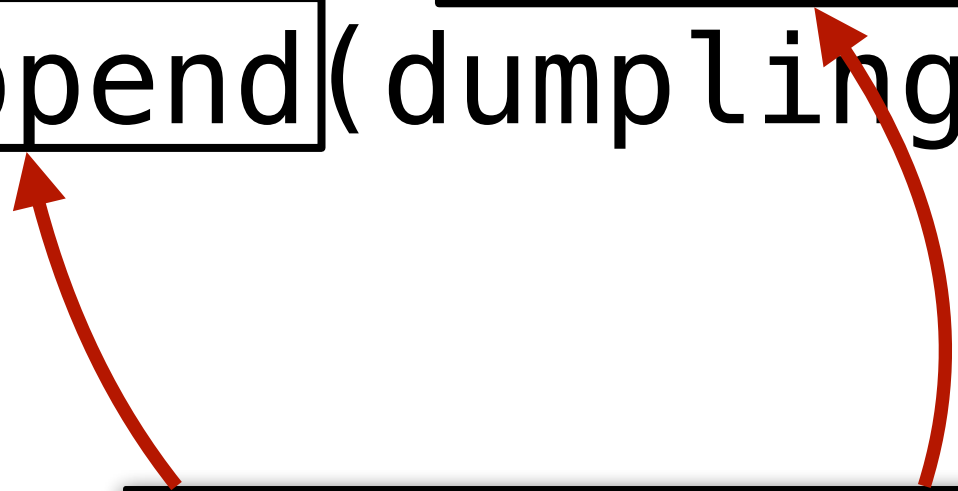
```
pancakes = [list: "egg", "butter", "flour",  
               "sugar", "salt", "baking powder", "blueberries"]  
dumplings = [list: "egg", "wonton wrappers",  
                  "pork", "garlic", "salt", "gf soy sauce"]  
pasta = [list: "spaghetti", "tomatoes",  
              "garlic", "onion"]
```

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))
```




*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)
```



***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"]
```



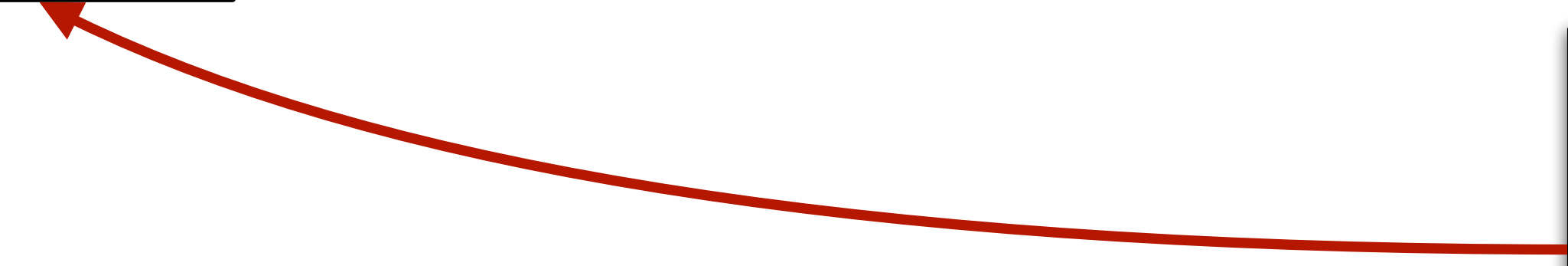
```
>>> shopping-list
```

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

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"]
```

This is an interesting new type annotation!

```
fun is-gluten-free(recipe :: List<String>) -> Boolean:
  doc: "Return true if none of the ingredients in a
  list contain gluten"
  L.length(
    L.filter(
      lam(i): L.member(gluten, i) end,
      recipe)) == 0
where:
  is-gluten-free(pancakes) is false
  is-gluten-free(dumplings) is true
end
```

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"
```

```
  L.length(  
    L.filter(  
      lam(i): L.member(gluten, i) end,  
      recipe)) == 0
```

How many elements are in the given list?

```
where:
```

```
  is-gluten-free(pancakes) is false
```

```
  is-gluten-free(dumplings) is true
```

```
end
```

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
```



*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
```


What if we want to take a recipe and make it vegan?

Sorry, meat-lovers!

Let's think about what the input and output should be. We're starting with the list of ingredients,

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

and it should become, say,

```
[list: "flax", "margarine", "flour",  
      "sugar", "salt", "baking powder",  
      "blueberries"]
```

We need an operation that produces a list, where some of the items are different than in the input list.

We can't do this with **member**, **distinct**, or **filter**.

L.map is similar to the **transform-column** function we used with tables.

It takes a function and a list as input and produces a list where each item is the result of running the function on the corresponding item of the input list.

```
fun veganize-ingredient(ingredient :: String) -> String:
  doc: "Change a non-vegan ingredient to its vegan
equivalent"
  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
end
```



```
fun veganize-recipe(recipe :: List<String>) -> List<String>:
  doc: "Update a recipe to be vegan"
  L.map(veganize-ingredient, recipe)
where:
  veganize-recipe(pasta) is pasta
  veganize-recipe(dumplings) is
    [list: "flax", "wonton wrappers",
      "mushroom", "garlic", "salt", "soy sauce"]
end
```

Because **veganize-ingredient** is just a helper function for **veganize-recipe**, we might prefer to define it inside **veganize-recipe**:

```
fun veganize-recipe(recipe :: List<String>) -> List<String>:
  fun veganize-ingredient(ingredient :: String) -> String:
    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
end
  L.map(veganize-ingredient, recipe)
where:
  veganize-recipe(pasta) is pasta
  veganize-recipe(dumplings) is [list: "flax", "wonton wrappers",
    "mushroom", "garlic", "salt", "soy sauce"]
end
```

Operation signatures

What operations have we seen so far?

L.member

List, *⟨item⟩* → Boolean

Indicates whether the item is in the list

L.distinct

List → List

Returns the unique values from input list

L.filter

Function, List → List

Returns list of items from input list on which function returns true
(in the same order as in the input list)

...

*Can we get
more specific?*

What operations have we seen so far?

L.member

List, *<item>* -> Boolean

Indicates whether the item is in the list

L.distinct

List -> List

Returns the unique values from input list

L.filter

Function, List -> List

Returns list of items from input list on which function returns true
(in the same order as in the input list)

...

What operations have we seen so far?

L.member

List, *<item>* -> Boolean

Indicates whether the item is in the list

L.distinct

List -> List

Returns the unique values from input list

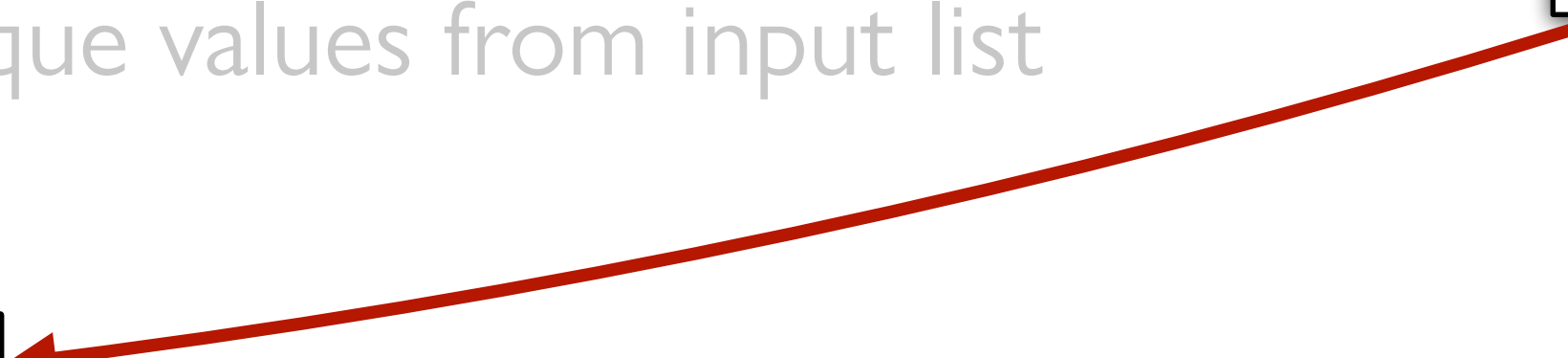
L.filter

(a -> Boolean), List -> List

Returns list of items from input list on which function returns true
(in the same order as in the input list)

...

*A function that takes an input of some type – call it *a* – and returns a Boolean.*



What operations have we seen so far?

L.member

List, *<item>* -> Boolean

Indicates whether the item is in the list

L.distinct

List -> List

Returns the unique values from input list

L.filter

(a -> Boolean), List<a> -> List<a>

Returns list of items from input list on which function returns true
(in the same order as in the input list)

...

The items in the input and output lists will be of the same type a, otherwise we couldn't run the predicate function on them.

What about **map**?

Function, List \rightarrow List

What about **map**?

(a -> b), List -> List

*A function that takes an input of some type – call it **a** – and returns an output of type **b**, which might be the same as **a** or might not. E.g., we might be taking a Number and converting it to a String.*

What about **map**?

$(a \rightarrow b), \text{List}\langle a \rangle \rightarrow \text{List}$

*The input list needs to be made of
as that we can give to that
function.*

What about **map**?

`(a -> b), List<a> -> List`

*The output list will be made of the
bs that the function returned.*

For a full list of operations and their signatures, see the [Pyret lists documentation](#).

Lists and tables

We've seen one way of describing a set of recipes – as a set of hardcoded lists.

This makes sense when we have a small set of recipes that doesn't change often, but we might want something better.

Another possibility would be to use a table with one column per ingredient:

```
recipes1 = table:  
  name :: String, spaghetti :: Boolean, milk :: Boolean,  
  tomatoes :: Boolean, onions :: Boolean, blueberries :: Boolean,  
  garlic :: Boolean, salt :: Boolean  
  row: "pasta", true, false, true, true, false, true, true  
end
```

name	spaghetti	milk	tomatoes	onions	blueberries	garlic	salt
"pasta"	true	false	true	true	false	true	true

The table would let us make plots and charts using the operations we know in Pyret

The lists are easier to write and modify

The tables could become sparse if we add more categories and ingredients

Whether you use tables or lists depends on the data you have and how you plan to use it.

For the programs we've written today, the lists were sufficient and lightweight, so they were the better choice.

Other programs might have benefitted from the table-shaped data.

Another possibility we'll return to later is combining lists and tables, e.g.,

```
recipes2 = table:  
  name :: String, ingredients :: List<String>  
  row: "pasta", [list: "spaghetti", "tomatoes", "garlic", "onion", "salt"]  
end
```

name	ingredients
"pasta"	[list: "spaghetti", "tomatoes", "garlic", "onion", "salt"]

