CMPU 101 § 1 · Problem-Solving and Abstraction

The Secret Nature of Lists

6 October 2021

Where are we?

We’ve been working with tables for the past few weeks.

Last class we saw a new data type: lists.

A *list* is like the column of a table – an ordered collection of items, possibly including duplicates.

```
table:
  number-grade, letter-grade
  row: 98, "A"
  row: 100, "A"
  row: 74, "C"
  row: 84, "B"
end
```

A *list* is like the column of a table – an ordered collection of items, possibly including duplicates.

number-grade	letter-grade
98	"A"
100	"A"
74	"C"
84	"B"

A *list* is like the column of a table – an ordered collection of items, possibly including duplicates.

number-grade	letter-grade
98	"A"
100	"A"
74	"C"
84	"B"

A *list* is like the column of a table – an ordered collection of items, possibly including duplicates.

number-grade	letter-grade
98	"A"
100	"A"
74	"C"
84	"B"

```
[list:
  "A",
  "A",
  "C",
  "B"]
```

Columns in a table can contain a mix of different data types, e.g.,

```
table: grades
  row: 98
  row: 56
  row: 74
  row: "F"
  row: "A"
  row: "B"
end
```

And so can a list:

```
[list: 98, 56, 74, "F", "A", "B"]
```

However, we usually find it easier to work with a column where every value is of the same kind.

We saw *data sanitizer* functions that can help us achieve this.

Additionally, we can *annotate* the type of data in the column when we make a table:

```
table: col :: Number
  row: 1
  row: 2
  row: 3
end
```

```
table: col :: String
  row: "a"
  row: "b"
  row: "c"
end
```

Likewise, we'll most often have just one type of data in a list, and we can change how we write the data type to show that.

For example,

```
[list: 1, 2, 3]
```

List<Number>

"list of numbers"

```
[list: "a", "b", "c"]
```

List<String>

"list of strings"

When we want to work with lists, we start by loading the functions for doing so, giving them a special prefix, **L**:

```
import lists as L
```

We saw some basic functions to ask questions about lists:

```
››› l = [list: "a", "b", "c"]
››› L.length(l)
3
››› L.member(l, "a")
true
››› L.member(l, "d")
false
```

When we wanted to put two lists together, we could append them like we did with strings:

```
››› l = [list: "a", "b", "c"]
››› m = [list: "d", "e", "f"]
››› L.append(l, m)
[list: "a", "b", "c", "d", "e", "f"]
```

And if we had a list with duplicate items, we could get a list of just the distinct items:

```
››› l = [list: "a", "b", "c"]
››› uh-oh = L.append(l, l)
››› uh-oh
[list: "a", "b", "c", "a", "b", "c"]
››› L.distinct(uh-oh)
[list: "a", "b", "c"]
```

Many of the functions we've used for working with tables have analogues that work with lists.

For instance, when we wanted just certain rows in a table, we used **`filter-with`**, giving it the table and a function that returned true for the rows we want to keep.

To do the same thing for items in a list, use **`L.filter`**.

```
››› l = [list: "a", "b", "c"]
››› L.filter(
      lam(i): not(i == "a") end,
      l)
[list: "b", "c"]
```

```
>>> l = [list: "a", "b", "c"]
>>> L.filter(
      lam(i): not(i == "a") end,
      l)
[list: "b", "c"]
```

This is an anonymous (i.e., unnamed) function made using a lambda expression.

One difference to be aware of:

```
filter-with(⟨table⟩, ⟨function⟩)

L.filter(⟨function⟩, ⟨list⟩)
```

When you're working with a list, the function argument comes first.

Often we use **`L.filter`** to ask the questions

"Does this function return `true` for *every* item in the list?"

```
L.length(L.filter(function, list)) == L.length(list)
```

"Does this function return `true` for *any* item in the list?"

```
L.length(L.filter(function, list)) > 0
```

Often we use **`L.filter`** to ask the questions

"Does this function return `true` for *every* item in the list?"

```
L.length(L.filter(function, list)) == L.length(list)
```

```
L.all(function, list)
```

"Does this function return `true` for *any* item in the list?"

```
L.length(L.filter(function, list)) > 0
```

Often we use **`L.filter`** to ask the questions

"Does this function return `true` for *every* item in the list?"

```
L.length(L.filter(function, list)) == L.length(list)
```

```
L.all(function, list)
```

"Does this function return `true` for *any* item in the list?"

```
L.length(L.filter(function, list)) > 0
```

```
L.any(function, list)
```

Data representation

Last class, we saw one way of describing a set of recipes – as a set of hardcoded lists:

```
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"]
```

This makes sense when we have a small set of recipes that doesn't change often.

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 what we‘ve written, 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"]

Transforming a list

In addition to the ingredients in the recipes, we encoded some dietary restrictions:

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

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

Let's think about what the input and output should be. We're starting with the recipe's 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 **`L.member`**, **`L.distinct`**, or **`L.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
```

List 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)

...

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

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

For a full list of operations and their signatures, see the Pyret lists documentation.

Designing list functions

How would we write a function that takes a list of numbers and returns its sum?

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
end
```

Your first thought should be whether you can write the body using one of the list functions we already know.

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
  my-sum([list: 1, 4]) is 1 + 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
  my-sum([list: 1, 4]) is 1 + 4
  my-sum([list: 4]) is 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
  my-sum([list: 1, 4]) is 1 + 4
  my-sum([list: 4]) is 4
  my-sum([list: ]) is ...
end
```

This is a little weird. There are no numbers left in this list.

We can have a string with no characters in it:

```
""
```

And, likewise, we can have a list with no items in it:

```
[list: ]
```

For these data types, these values are the equivalent of 0, the number representing no quantity.

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
  my-sum([list: 1, 4]) is 1 + 4
  my-sum([list: 4]) is 4
  my-sum([list: ]) is 0
end
```

If there are no numbers, their sum must be 0!

```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```

The secret nature of lists

Writing our input as `[list: 3, 1, 4]` is a lie.

It's just a shorthand for the real structure of a list.

In its secret heart, Pyret knows there are only two ways of making a list:

There's the empty list, **`empty`**.

And there's adding an element to the front of an existing list using the **`link`** function.

When we write an expression like

```
[list: 1, 2, 3],
```

Pyret translates it for us into a composition of these possibilities:

```
link(3,
  link(1,
    link(4, empty)))
```

Using the secret

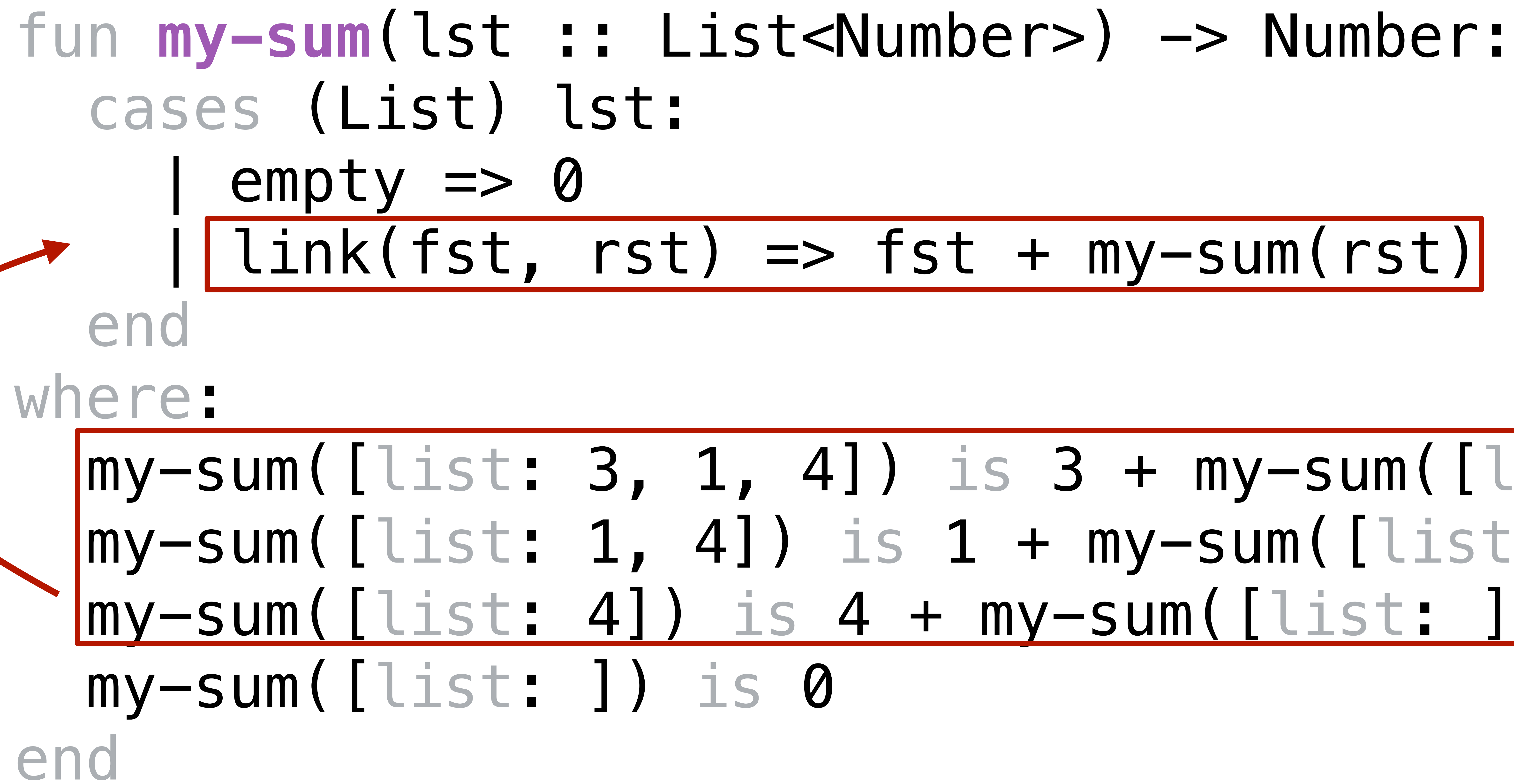
```
fun my-sum(lst :: List<Number>) -> Number:
  ...
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => ...
    | link(fst, rst) => ...
  end
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```

A `cases` expression is like an `if`. If the list is `empty`, do one thing. If it's a `link`, do another thing.

```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => ...
    | link(fst, rst) => ...
  end
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```

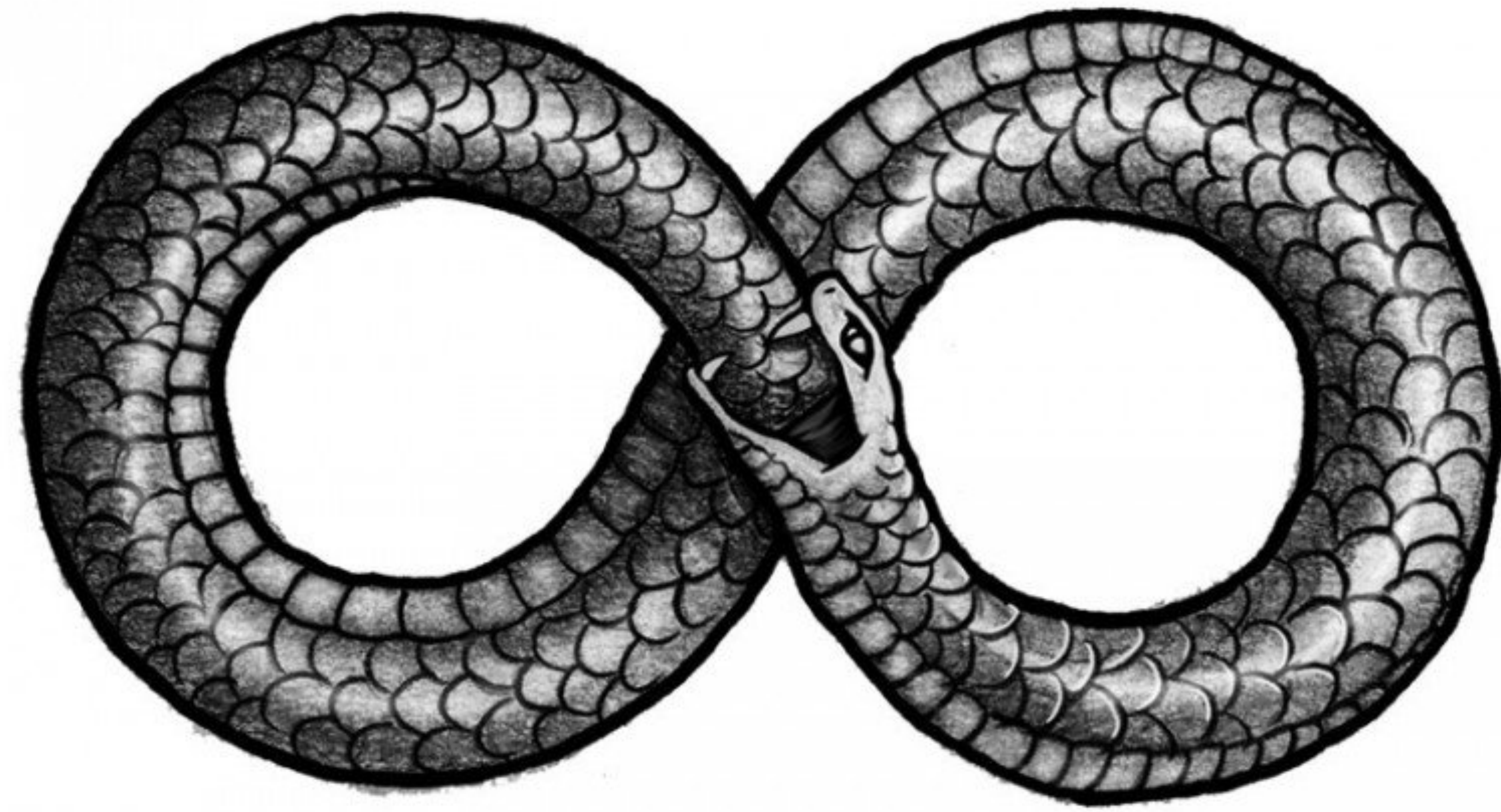
```
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(fst, rst) => ...
  end
where:
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: ]) is 0
end
```


fun my-sum(lst :: List<Number>) -> Number:
cases (List) lst:
| empty => 0
| link(fst, rst) => fst + my-sum(rst)
end
where:
my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
my-sum([list: 4]) is 4 + my-sum([list:])
my-sum([list:]) is 0
end

When we call this function, it evaluates as:

```
my-sum(link(3, link(1, link(4, empty))))
3 + my-sum(link(1, link(4, empty)))
3 + 1 + my-sum(link(4, empty))
3 + 1 + 4 + my-sum(empty)
3 + 1 + 4 + 0
```

When we call **`my-sum`**, the result it returns depends on other calls to **`my-sum`**!



Lecture code:

`https://code.pyret.org/editor#share=1cMg6RAjFa9dK4-OQCmLl3Dhktpne1qS0&v=1904b2c`

Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University

Doug Woos, Brown University