CMPU 101 § 53 · Computer Science I

Data Definitions

13 February 2024

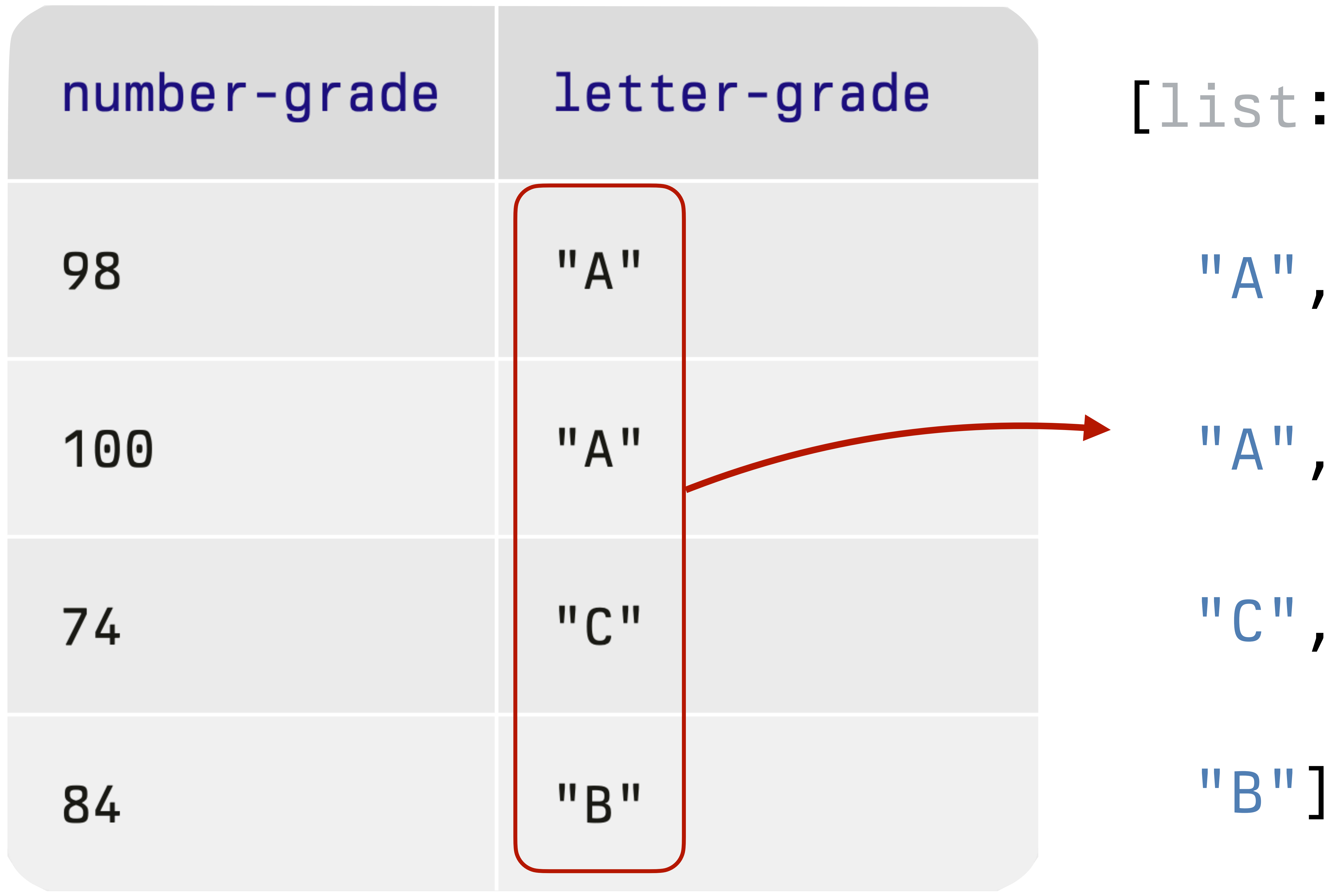
VASSAR COLLEGE
A·D·1861

Where are we?

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

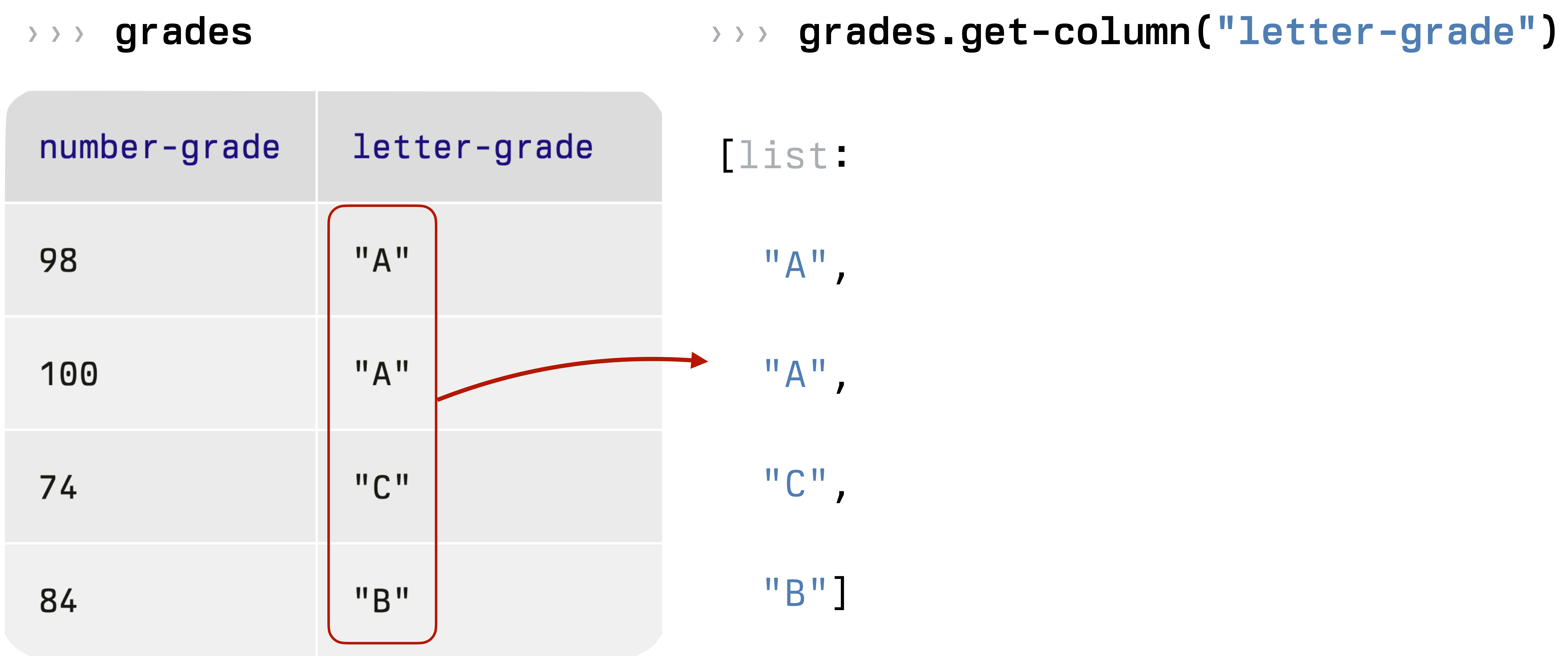
Last class we saw a new data type: lists.

```
››› grades
```



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

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

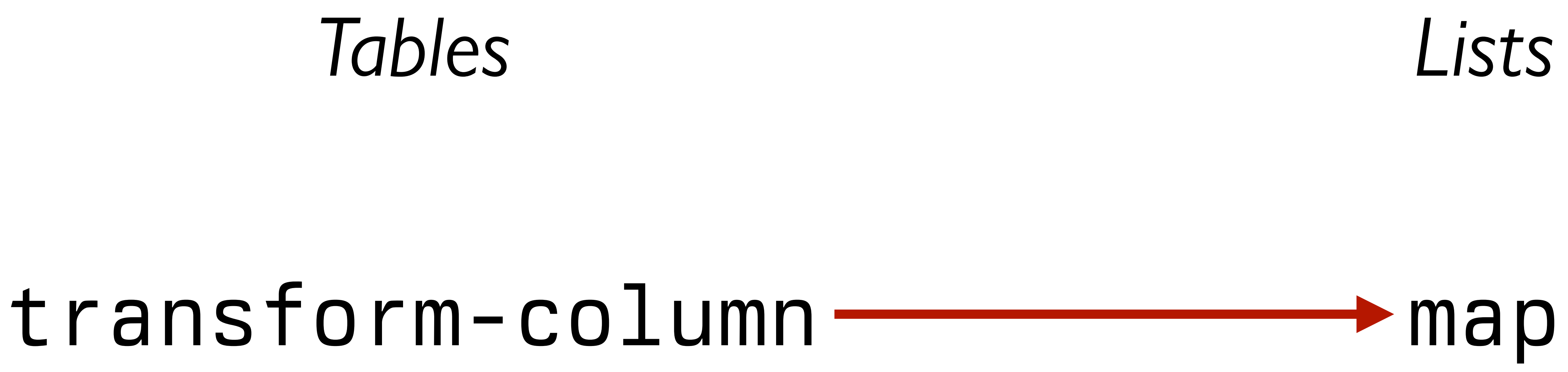


```
››› grades
```

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

```
››› grades.get-column("letter-grade")

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

We used higher-order functions to work with tables, and we can do the same with lists:



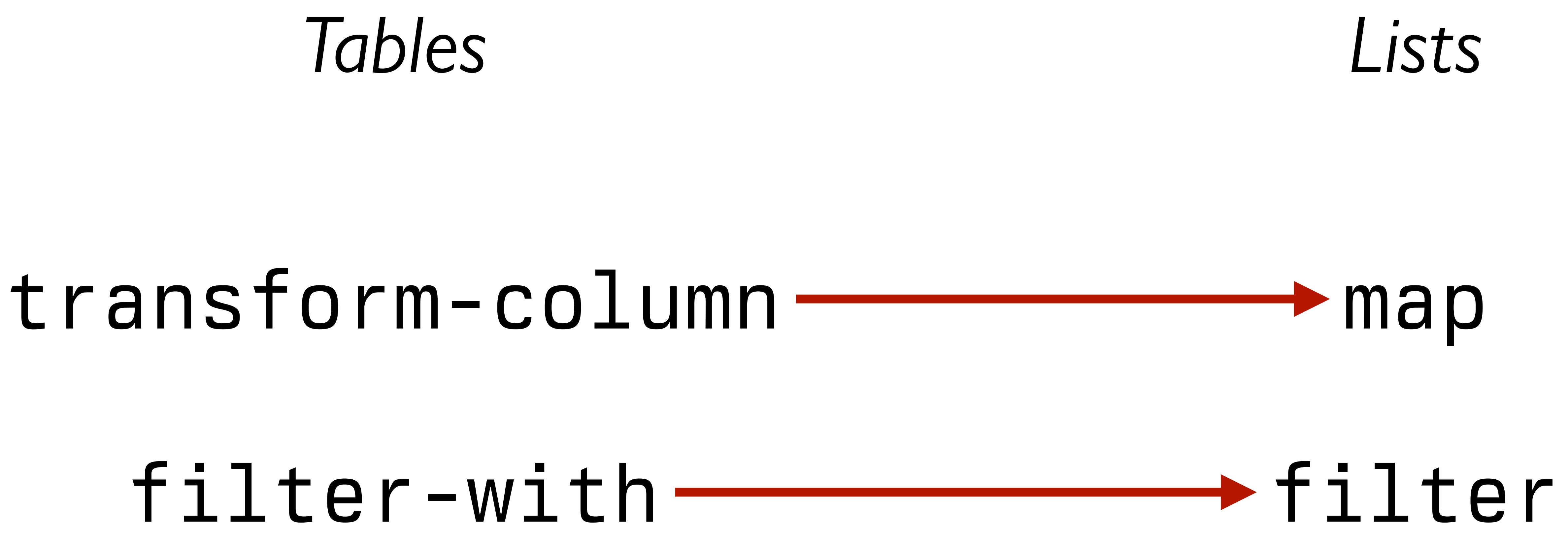
Tables

Lists

`transform-column`

`map`

We used higher-order functions to work with tables, and we can do the same with lists:



Tables	*Lists*
`transform-column`	`map`
`filter-with`	`filter`

```
››› animals = [list: "bear", "cat", "dog"]
››› filter(lam(a): a <> "bear" end,
      animals)
[list: "cat", "dog"]
```

```
››› animals = [list: "bear", "cat", "dog"]
››› filter(lam(a): a <> "bear" end,
      animals)
[list: "cat", "dog"]
```

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

Numbers, strings, images, Booleans, tables, and lists let us represent many kinds of real data quite naturally.

But there are times when we're going to want something a bit different.

Defining structured data

Imagine that we're doing a study on communication patterns among students.

We don't have access to the messages the students sent – hopefully they're encrypted! – but we have *metadata* for each message:

sender

recipient

day of the week

time (hour and minute)

This kind of metadata might sound uninteresting, but it can tell us a lot!

’re doing a study on communication
students.

We don’t have access to the messages the students sent – hopefully they’re encrypted! – but we have *metadata* for each message:

- sender
- recipient
- day of the week
- time (hour and minute)

This kind of metadata might sound uninteresting, but it can tell us a lot!

Recommended reading:

John Bohannon, “Your call and text records are far more revealing than you think”, *Science*, 2016

’re doing a

students.

We don’t have access to the messages the students sent – hopefully they’re encrypted! – but we have *metadata* for each message:

sender

recipient

day of the week

time (hour and minute)

Imagine that we’re doing a study on communication patterns among students.

We don’t have access to the messages the students sent – hopefully they’re encrypted! – but we have *metadata* for each message:

sender

recipient

day of the week

time (hour and minute)

Imagine that we're doing a study on communication patterns among students.

We don't have access to the messages the students sent – hopefully they're encrypted! – but we have *metadata* for each message:

sender

recipient

day of the week

time (hour and minute)

How should we store this data?

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time ::* …
"4015551234"	"8025551234"	"Mon"	…

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: String*
`"4015551234"`	`"8025551234"`	`"Mon"`	`"4:55"`

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: Number*
"4015551234"	"8025551234"	"Mon"	295

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: List*
`"4015551234"`	`"8025551234"`	`"Mon"`	`[list: 4, 55]`

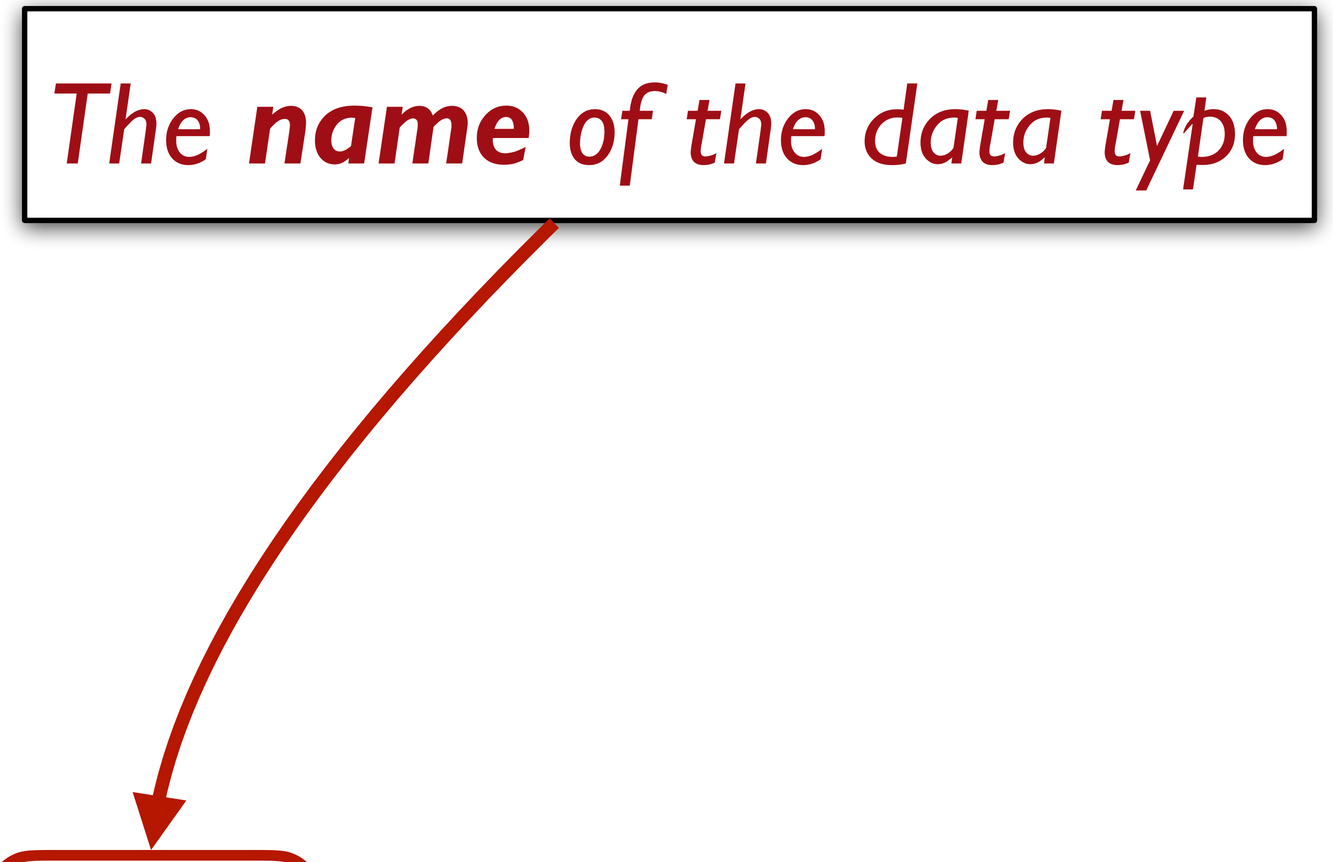
We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*hour :: Number*	*minute :: Number*
`"4015551234"`	`"8025551234"`	`"Mon"`	4	55

If we use multiple columns, we can access the components independently, by name, but if we use a single column, all of the “time” data is in one place.

To resolve this trade-off, we add structure: We can have a single data type that has named parts.

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```



*The **name** of the data type*

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

*A **constructor** function that builds the data type*

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

*The **components** of the data*

After defining the data type,

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

we can call **`time`** to build **`Time`** values,

```
››› noon = time(12, 0)
››› half-past-three = time(3, 30)
```

and we can use dot notation to access the components:

```
››› noon.hours
12
››› half-past-three.mins
30
```

Our table could now be:

sender :: String	*recipient :: String*	*day :: String*	*time :: Time*
`"4015551234"`	`"8025551234"`	`"Mon"`	`time(4, 55)`

Conditional data

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

The only way to make a **`Time`** *is to call the* **`time()`** *constructor function.*

But we can also define *conditional data*, where there are multiple varieties of the data.

The varieties can just be fixed values, e.g.,

```
data Day:
  | sunday
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
end
```

Or they can be separate constructors, e.g.,

```
data Message:
  | direct(sender :: String,
      recipient :: String,
      message :: String)
  | group(sender :: String,
      recipients :: List<String>,
      message :: String)
end
```

Or we can mix these together, e.g.,

```
data Name:
  | name(first :: String, last :: String)
  | anonymous
end
```

Recursive data definitions

Last week we worked with *lists* – ordered sequences of items, equivalent to a column in a table.

Much like the rows in a table, the items in a list have numeric indices:

```
                  0    1    2
››› lst = [list: "a", "b", "c"]
```

And we can access items using these indices:

```
››› lst.get(0)
"a"
››› lst.get(1)
"b"
```

But writing the list as `[list: "a", "b", "c"]` is just a convenient deception!

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

A list is either:

empty or

linking an item to another list.

That is, a list is a kind of conditional data:

```
data List:
  | empty
  | link(first :: Any, rest :: List)
end
```

That is, a list is a kind of conditional data:

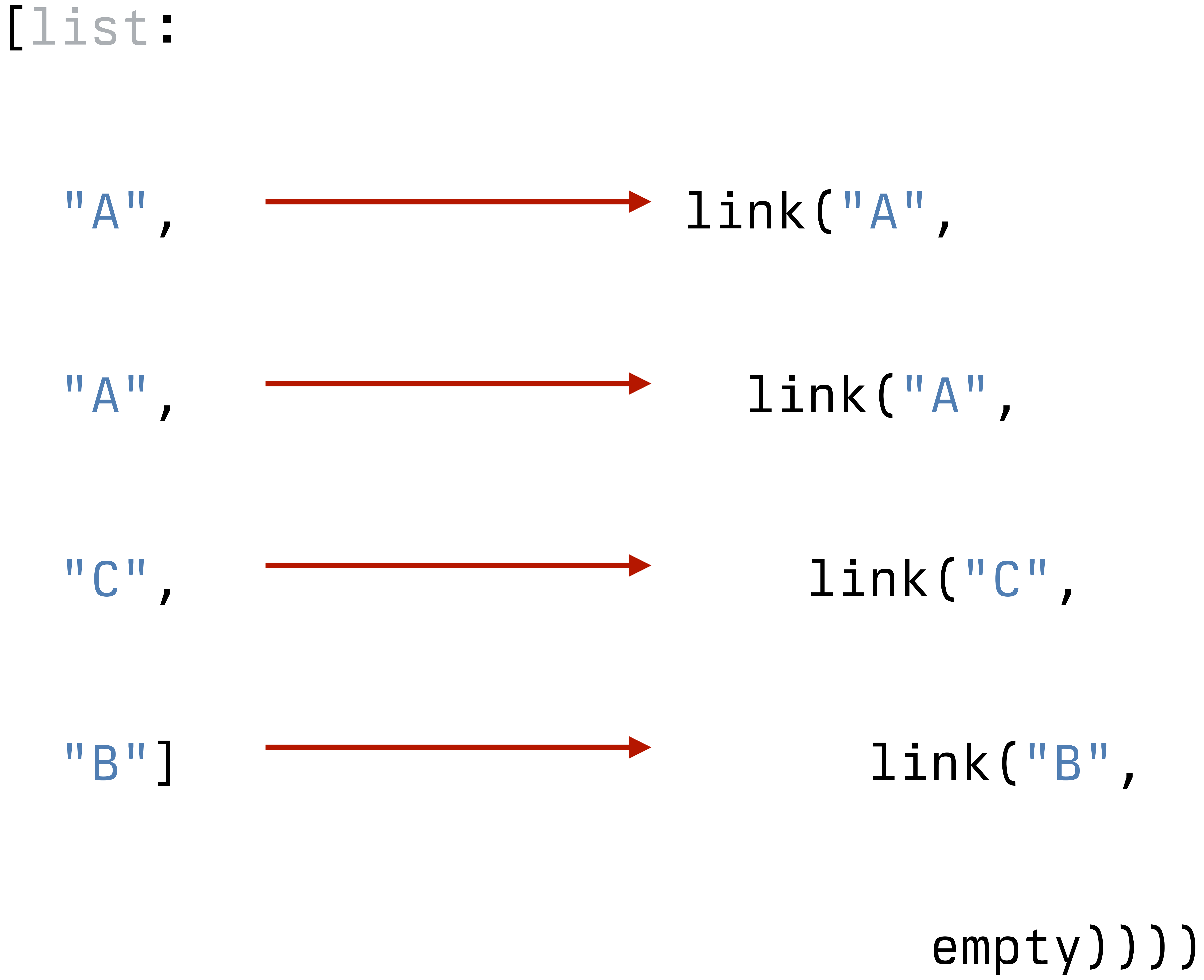
```
data List:
  | empty
  | link(first :: Any, rest :: List)
end
```

So, a list of one item, e.g.,

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

is really a link between an item and the empty list:

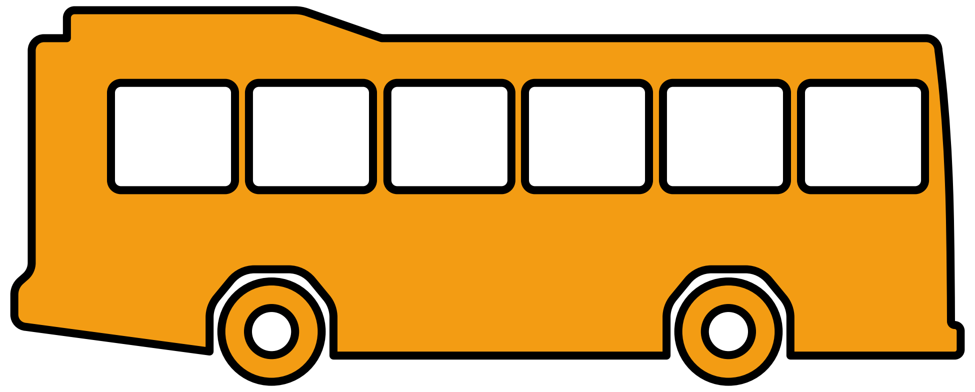
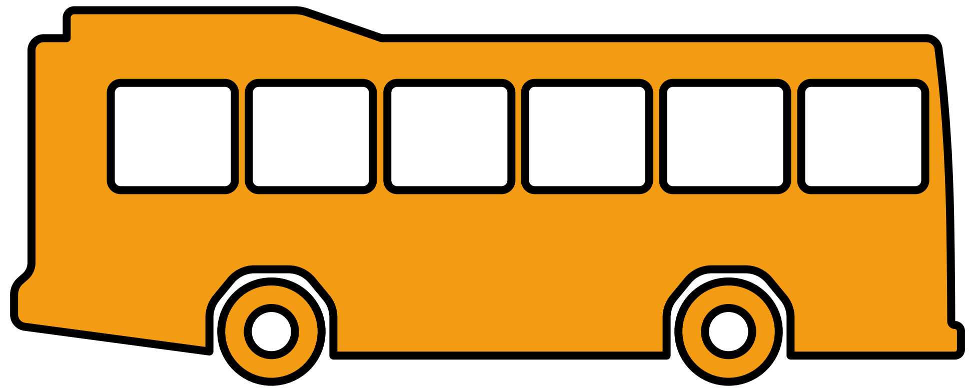
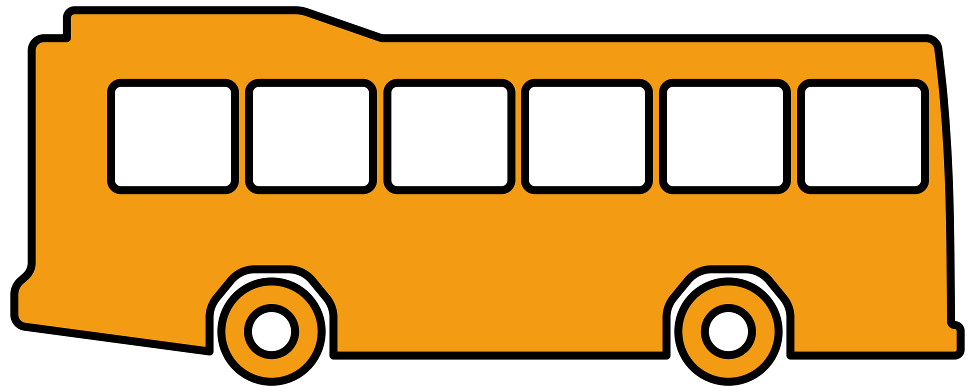
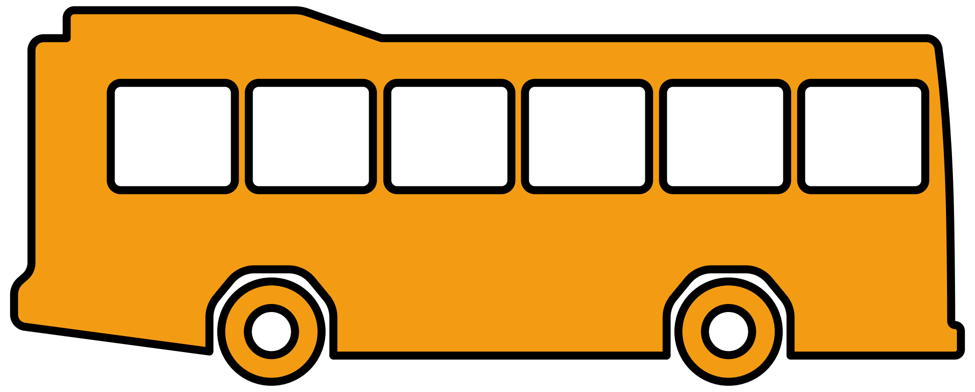
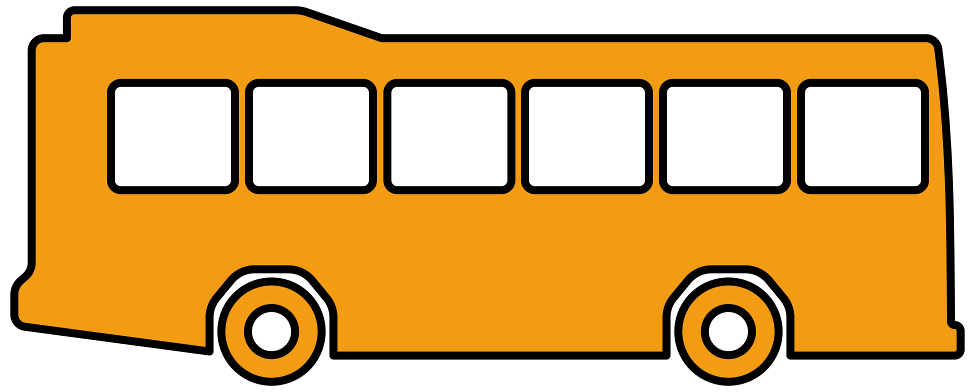
```
link("A", empty)
```



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

```
link("A",
  link("A",
    link("C",
      link("B",
        empty))))
```

Recursion

1

1+1=2
1

2+1=3
1+1=2
1

3+1=4
2+1=3
1+1=2
1

4+1=5
3+1=4
2+1=3
1+1=2
1

4+1=5
3+1=4
2+1=3
1+1=2
1

Count all the buses

4+1=5
3+1=4
2+1=3
1+1=2
1

Count all the buses

Count all the buses

4+1=5
3+1=4
2+1=3
1+1=2
1

Count all the buses

Count all the buses

Count all the buses

4+1=5
3+1=4
2+1=3
1+1=2
1

Count all the buses

Count all the buses

Count all the buses

Count all the buses

4+1=5
3+1=4
2+1=3
1+1=2
1

Count one bus

Count all the buses

Count all the buses

Count all the buses

Count all the buses

Recursion is a programming technique where a problem is solved by solving a smaller version of the same problem, unless that smaller version is simple enough to solve directly.

We call the small version that can be solved directly the *base case* of the recursive problem.

To write our own functions to process a list, item by item, we need to think recursively, using the data definition of a list.

Designing functions using the definition of a list

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

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is ...
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4
  my-sum([list: 1, 4]) is 1 + 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4
  my-sum([list: 1, 4]) is 1 + 4
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list:        ]) is             0
  my-sum([list:       4]) is         4
  my-sum([list:    1, 4]) is     1 + 4
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list:        ]) is             0
  my-sum([list:       4]) is         4 + 0
  my-sum([list:    1, 4]) is     1 + 4 + 0
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4 + 0
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list:        ]) is             0
  my-sum([list:       4]) is         4 + my-sum([list: ])
  my-sum([list:    1, 4]) is     1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

*`cases` is like a special `if` expression that we use to ask “which **shape** of data do I have?”*

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

If the list is `empty`, do one thing.

If it’s a `link`, do another thing.

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

Denotes the output of a function

Marks the expression to evaluate if the data has the shape on the left.

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

This gives names for referring to the arguments to `my-sum`*.*

And this is giving names for referring to the arguments to `link`*.*

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      0

    | link(f, r) =>
      ...

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      0

    | link(f, r) =>
      f + my-sum(r)

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      0

    | link(f, r) =>
      f + my-sum(r)

  end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  end
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
```

```
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  end
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
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
```

Thinking recursively

Any time a problem is structured such that the solution on larger inputs can be built from the solution on smaller inputs, recursion is appropriate.

All recursive functions have these two parts:

Base case(s):

What’s the simplest case to solve?

Recursive case(s):

What’s the relationship between the current case and the answer to a slightly smaller case?

You should be calling the function you’re defining here; this is referred to as a *recursive call*.

```
fun recursive-function(lst :: List) -> ...:
  cases (List) lst:
    | empty =>
      ...

    | link(f, r) =>
      ... recursive-function(r) ...

  end
end
```

Base case

Recursive case

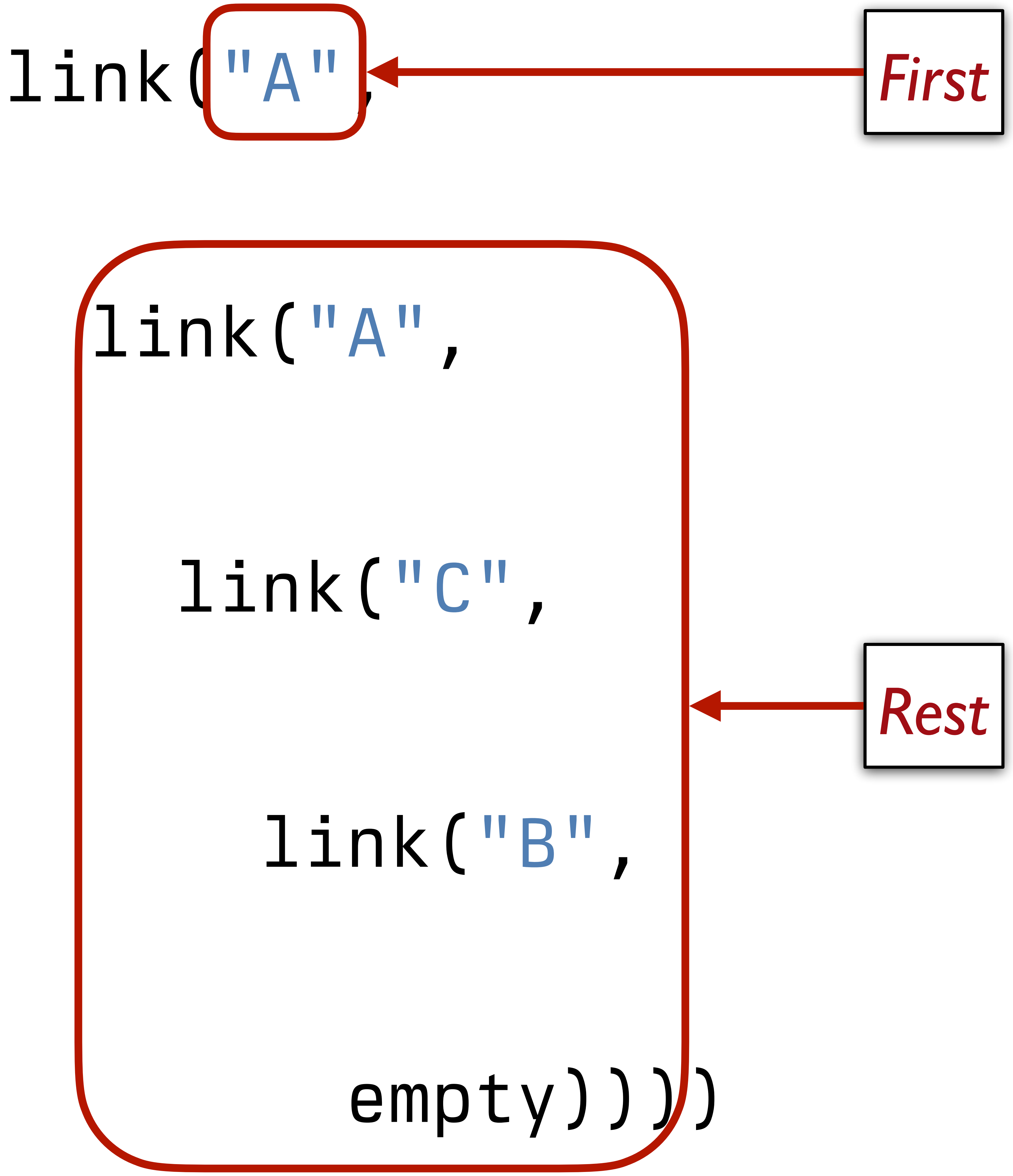
Each time you make a recursive call, you must make the input smaller somehow.

- If your input is a list, you pass the ***rest*** of the list to the recursive call.

```
link("A",
  link("A",
    link("C",
      link("B",
        empty))))
```

```
link("A",
 link("A",
  link("C",
   link("B",
    empty))))
```

First



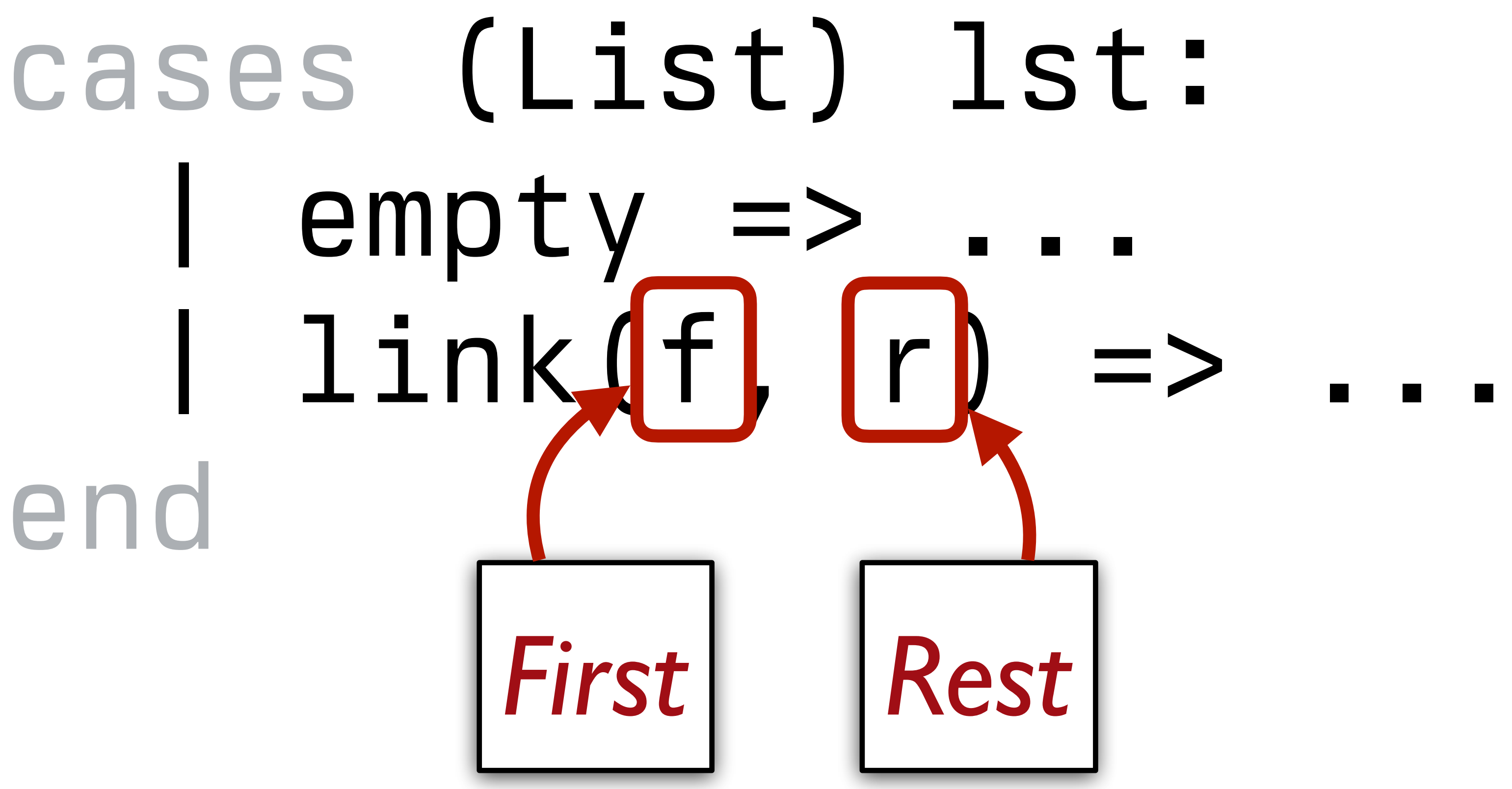
```
link("A",
```

```
link("A",

  link("C",

    link("B",

      empty))))
```

First

Rest

```
››› lst = [list: "item 1", "and", "so", "on"]
››› lst.first
"item 1"
››› lst.rest
[list: "and", "so", "on"]
```



```
cases (List) lst:
  | empty => ...
  | link(f, r) => ...
end
```

First

Rest

What happens if we *don’t* make the input smaller?

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

Recursive call on the rest of the input list

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

Recursive call on the original input list

When we call this function, it evaluates as:

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

This isn’t going to end well.

When a recursive function never stops calling itself, it's called *infinite recursion*.

Final note

Lists, recursion, and **`cases`** syntax are not easy concepts to grasp separately, much less all together in a short time.

Don’t feel frustrated if it takes a little while for these to make sense. Give yourself time, be sure to practice working in Pyret, and ask questions.

Class code:

`tinyurl.com/101-2024-02-13`

Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University

Ab Mosca, Northeastern University

Doug Woos, Brown University