Data Definitions

13 February 2024
Where are we?
We’ve been working with tables for the past few weeks.

Last class we saw a new data type: lists.
<table>
<thead>
<tr>
<th>number-grade</th>
<th>letter-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>100</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>74</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>84</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>

```python
>>> grades

["A",
 "A",
 "A",
 "C",
 "B"]
```
grades

<table>
<thead>
<tr>
<th>number-grade</th>
<th>letter-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>100</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>74</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>84</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>

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

["A", "A", "C", "B"]
We used higher-order functions to work with tables, and we can do the same with lists:

`transform-column` → `map`

**Tables**

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

\[\text{Tables} \quad \text{Lists}\]

- transform-column \rightarrow map
- filter-with \rightarrow 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)
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)

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
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.,

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>time :: …</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>…</td>
</tr>
</tbody>
</table>
We could have a table, e.g.,

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>time :: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>&quot;4:55&quot;</td>
</tr>
</tbody>
</table>
We could have a table, e.g.,

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>time :: Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>295</td>
</tr>
</tbody>
</table>
We could have a table, e.g.,

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>time :: List</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>[list: 4, 55]</td>
</tr>
</tbody>
</table>
We could have a table, e.g.,

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>hour :: Number</th>
<th>minute :: Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>4</td>
<td>55</td>
</tr>
</tbody>
</table>
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,

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

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

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

and we can use dot notation to access the components:

```plaintext
>>> noon.hours
12
>>> half-past-three.mins
30
```
Our table could now be:

<table>
<thead>
<tr>
<th>sender :: String</th>
<th>recipient :: String</th>
<th>day :: String</th>
<th>time :: Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;4015551234&quot;</td>
<td>&quot;8025551234&quot;</td>
<td>&quot;Mon&quot;</td>
<td>time(4, 55)</td>
</tr>
</tbody>
</table>
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.,

```kotlin
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.,

```java
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:

```python
>>> lst = ['a', 'b', 'c']
```

And we can access items using these indices:

```python
>>> lst[0]
'a'
>>> lst[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:

```plaintext
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.,

\[\text{list: } "A"\],

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

\text{link("A", empty)}
[list:

"A",

"A",

"C",

"B"]

link("A",

link("A",

link("C",

link("B",

empty)))]
Recursion
1+1=2
2+1=3
1+1=2
1 + 1 = 2
3 + 1 = 4
2 + 1 = 3
1 + 1 = 2
4+1=5
3+1=4
2+1=3
1+1=2
4+1=5
3+1=4
2+1=3
1+1=2

Count all the buses
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5

Count all the buses
4 + 1 = 5
3 + 1 = 4
2 + 1 = 3
1 + 1 = 2

Count all the buses
1+1=2
2+1=3
3+1=4
4+1=5

Count all the buses

Count one bus
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 =>
      ...  
      If the list is empty, do one thing.
    | link(f, r) =>
      ...
      If it's a link, do another thing.
  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
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 =>
   | 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:

\[
\text{my-sum(link(3, link(1, link(4, empty)))))} \\
\rightarrow 3 + \text{my-sum(link(1, link(4, empty))}) \\
\rightarrow 3 + 1 + \text{my-sum(link(4, empty))} \\
\rightarrow 3 + 1 + 4 + \text{my-sum(empty)} \\
\rightarrow 3 + 1 + 4 + 0
\]
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 \texttt{recursive-function} (lst :: List) -> ... : 

\begin{itemize}
  \item \textbf{Base case} \hfill | empty => \ldots \hfill \textbf{Recursive case} \hfill | link(f, r) => \ldots \texttt{recursive-function}(r) \ldots
\end{itemize}
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)))))
link("A",
    link("C",
        link("B",
            empty))))

First

Rest
```python
>>> 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
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
When we call this function, it evaluates as:

\[
\text{my-sum(link(3, link(1, link(4, empty))))}
\]

\[
\rightarrow 3 + \text{my-sum(link(3, link(1, link(4, empty))))}
\]

\[
\rightarrow 3 + 3 + \text{my-sum(link(3, link(1, link(4, empty))))}
\]

\[
\rightarrow 3 + 3 + 3 + \text{my-sum(link(3, link(1, link(4, empty))))}
\]

... 

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