Designing Data Types

25 October 2021
Where are we?
We’ve seen numbers, strings, images, Booleans, tables, and lists.

These let us represent many kinds of real data quite naturally. But there are times when we’ll want something a bit different.
Defining structured data
Imagine that we’re doing a study on communication patterns between students.

While we don’t have the messages the students sent, we have the metadata for each message:

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

This is a realistic example, and this information is more important than it might seem!

You may want to read

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

science.org/content/article/your-call-and-text-records-are-far-more-revealing-you-think
Imagine that we’re doing a study on communication patterns between students.

While we don’t have the messages the students sent, we have the 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
data

| time(hours :: Number, mins :: Number)

end

*The name of the data type*
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,

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

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

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

and we can use dot notation to access the components:

```python
>>> noon.hours
12
>>> half-past.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>
And we can write functions that use the hour and minute components, e.g., `message-before` that takes a row (representing a message) and returns true if the message was sent before the specified time.
Code for **message-before**:

https://code.pyret.org/editor#share=1LB4oJC68oWrf7tLFr0BYabhtmdtDHUUK&v=1904b2c
Defining conditional data
There are many applications where we need to represent times, and we can reuse our `Time` data definition.

For example, if we want to build a calendar, that’s a collection of appointments, each of which has a

- Date
- Start time
- Duration
- Description
One possible design:

```haskell
data Date:
  | date(year :: Number, month :: Number,
      day :: Number)
end

data Event:
  | event(date :: Date, time :: Time,
      duration :: Number, descr :: String)
end

calendar :: List<Event> = …
```
Many calendar programs also offer a way to manage your to-do list.

Let’s say a to-do item has the following data:

- Task
- Deadline
- Urgency
We could have one list for calendar events and one for to-do items, but then we lose the benefit of having a single calendar with all our entries.

For many tasks (e.g., displaying entries sorted by date), we want both calendar events and to-do items.
Instead, we can define a *conditional data* type with multiple constructors:

```haskell
data Event:
  | appt(date :: Date, time :: Time, 
        duration :: Number, descr :: String)
  | todo(deadline :: Date, task :: String, 
        urgency :: String)
end
```
Now a calendar can be a `List<Event>`, containing both types of events, e.g.,

```haskell
calendar :: List<Event> =
  [list:
    appt(date(2021, 10, 25), time(13, 30), 75, "CMPU 101"),
    todo(date(2021, 10, 27),
         "Use avocado", "high")]
```
But how do we work with a list where the items can have different parts?

Well, we’ve already seen the way to work with different varieties of data; it’s *cases*. 
For example, if we want to search our calendar for all events related to a term, we could write a function `event-matches`.
Code for **event-matches**:  
https://code.pyret.org/editor#share=1LB4oJC68oWrf7tLFr0BYabhtmdtDHUUk&v=1904b2c
And we can use it to filter our calendar:

```plaintext
fun search-calendar(cal :: List<Event>,
                   term :: String) -> List<Event>:
    L.filter(lam(e): event-matches(e, term) end, cal)
end
```
Defining recursive data
A list is just a built-in kind of conditional data!

We used **cases** to tell apart its two possibilities – **empty** or **link**.
Now we can see how lists are defined:

```
data MyList:
    | my-empty
    | my-link(first, rest :: MyList)
end
```
Now we can see how lists are defined:

data \texttt{MyList}:
  | my-empty
  | my-link(first, rest :: MyList)
end
my-empty

my-link(1,
    my-link(2,
        my-link(3,
            my-empty))))
And just like we did for a List, we use this template to write a function that recursively processes the data:

```pascal
fun my-list-fun(ml :: MyList) -> ...:
  doc: "Template for a fn that takes a MyList"
  cases (MyList) ml:
    | my-empty => ...
    | my-link(f, r) =>
      ... f ...
      ... my-list-fun(r) ...
  end
  where:
    my-list-fun(...) is ...
end
```
Every data definition has a corresponding template. The more complex the data definition is – lots of variants, recursion, etc. – the more helpful it is to use the template!
Given a (recursive) data definition, you write a template by:

1. Creating a function header
2. Using cases to break the data input into its variants
3. In each case, listing each of the fields in the answer
4. Calling the function itself on any recursive fields
There’s no need to define MyList when we already have List, but next class we’ll see how the same idea of defining a recursive data type let’s us create something new!
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Doug Woos, Brown University