Designing new data types (structures)

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski

Data Types

- We've seen: Basic/Simple
  - Boolean
  - Number
  - String
- And: More Complex
  - Image
  - Table
  - List
- These data types may not be enough to suite our needs…
  - we must create them ourselves
  - These are called structures (struct data type in C, and similar to class in C++, Java)

Presented for your consideration

- We’re doing a study on communication patterns among students.
- We *don’t* have the messages the students sent,
- We do have the *metadata* for each message:
  - sender
  - recipient
  - day of the week
  - time (hour and minute)

Definition*: metadata is data that provides information *about* other data.

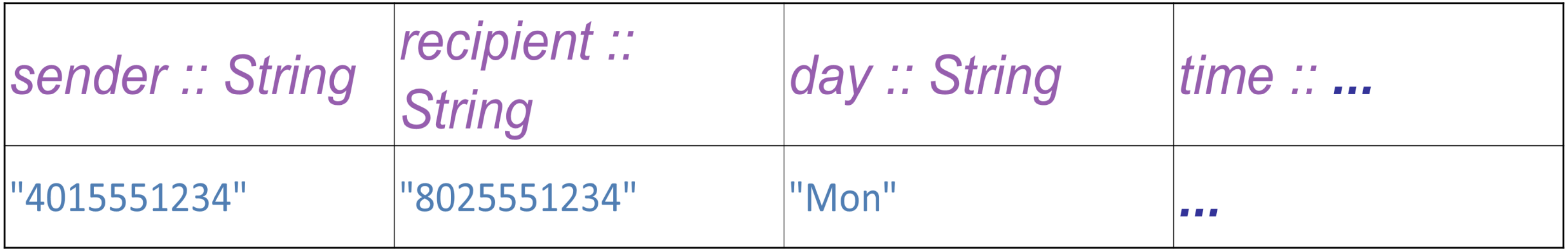
- *according to wikipedia
- A data type is one example

Text/Phone Call Metadata

- The NSA collects this metadata
  - For "national security" purposes
- See John Bohannon, "Your call and text records are far more revealing than you think", *Science*, 2016

How Should We Assemble Text Metdata?

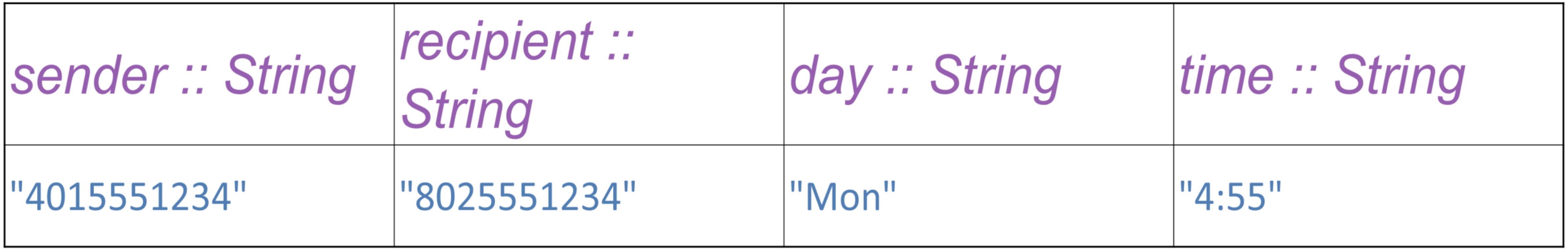
- The data suggests… a table!



sender :: String	recipient :: String	day :: String	time :: ...
"4015551234"	"8025551234"	"Mon"	...

How Should We Represent Time data

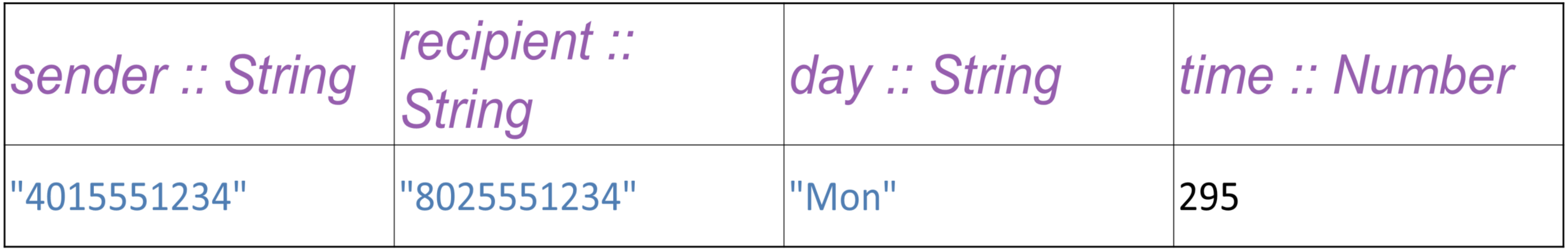
- A string?



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

How Should We Represent Time data

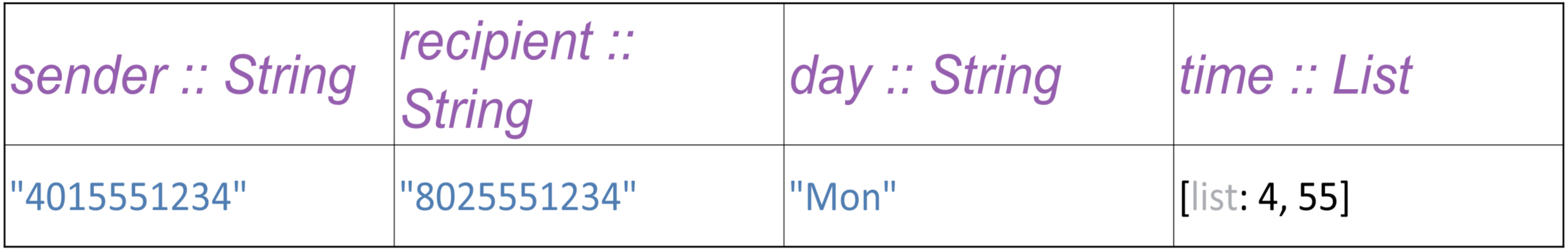
- A number, like the number of minutes since midnight?



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

How Should We Represent Time data

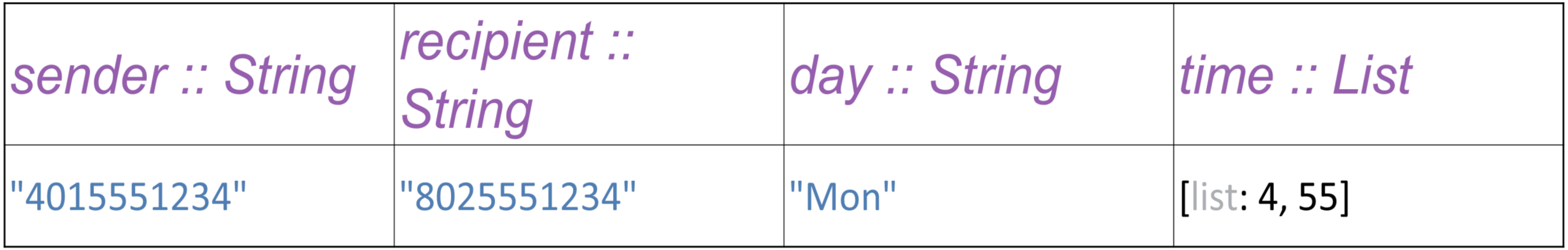
- A list?
  - Lists tend to be unbounded
  - Time requires exactly 2 entries



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

How Should We Represent Time data

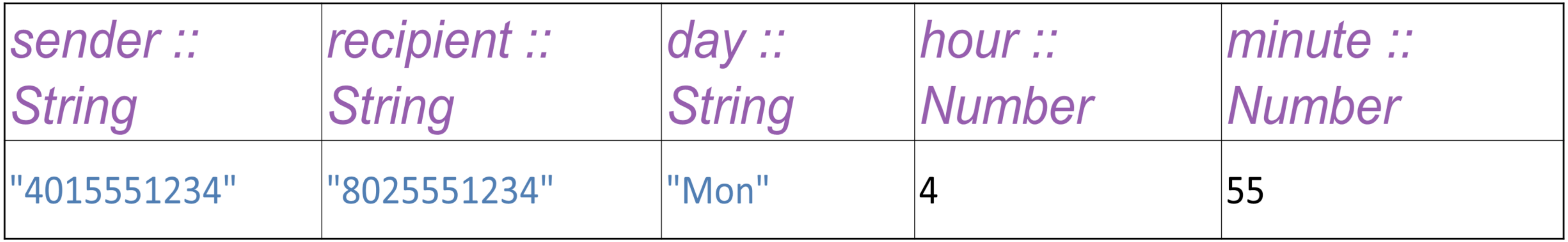
- A list?
  - The time is in one column, easy to read/access
  - Lists tend to be unbounded…
  - Whilst time requires exactly 2 entries



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

How Should We Represent Time data

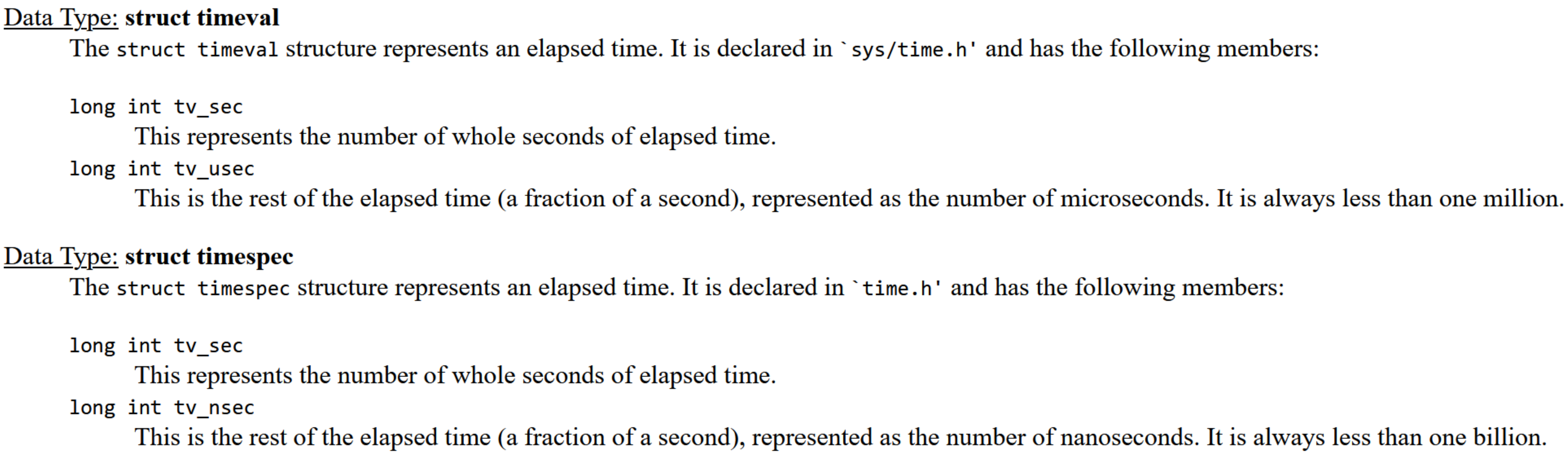
- A separate column
  - For hours and minutes?
    - We can access each number by name (!)



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

How GNU represents time data

- For C/C++, via GNU manual



Data Type: **struct timeval**

The `struct timeval` structure represents an elapsed time. It is declared in `` `sys/time.h' `` and has the following members:

`long int tv_sec`

This represents the number of whole seconds of elapsed time.

`long int tv_usec`

This is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million.

Data Type: **struct timespec**

The `struct timespec` structure represents an elapsed time. It is declared in `` `time.h' `` and has the following members:

`long int tv_sec`

This represents the number of whole seconds of elapsed time.

`long int tv_nsec`

This is the rest of the elapsed time (a fraction of a second), represented as the number of nanoseconds. It is always less than one billion.

Our Time Structure

- Provides both:
  - Easy access aspect of a list along with…
  - Individual names provided by separate columns

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

Our Time Structure (2)

```
# we define our own data type, named Time

data Time:

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

end
```

Our Time Structure (3)

```
# we define our own data type, named Time

data Time:

#we specify the makeup of time – a way to initialize or construct time.

#Then specify the named components of time (include the data type of each)

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

end
```

Using Our Time Structure

- #After defining the data type,

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

#we can call **time** to create an instance of **Time (note: Capital T!)** along with initial values,

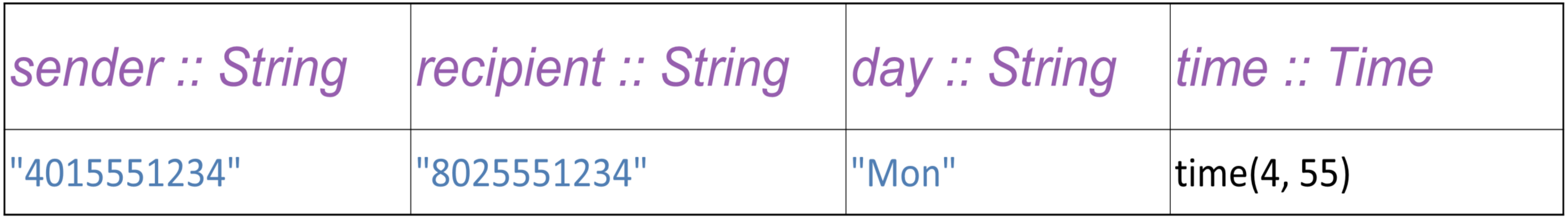
```
››› noon = time(12, 0)
››› half-past-three = time(3, 30)
```

#and we can use dot notation to access the components:

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

A new representation of Metadata

- Using our new data type, Time



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

Time analysis

- We can now write function to analyze our time data:
  - Let’s view this in pyret…

- **message-before** takes a row (representing a message) and returns true if the message was sent before the specified time.

Building A Better(?) Calendar

- If we want to build a calendar, a collection of appointments, each of which has a
  - Date
  - Start time
  - Duration
  - Description

Building A Better(!) Calendar

```
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> = ...
```

To-Do List

- Let’s say a to-do item has the following data:
  - Task
  - Deadline
  - Urgency/Priority

- For many tasks (e.g., displaying entries sorted by date), we want both calendar events and to-do items.
  - Let’s consider a “to-do” as another kind of event.

Conditional Data Type

We can define an *Event* data type with multiple constructors:

```
data Event:
  | appt(date :: Date, time :: Time,
     duration :: Number, descr :: String)
  | todo(deadline :: Date, task :: String,
     urgency :: String)
end
```

Conditional Data Type

We can define an *Event* data type with multiple constructors: one “stick key” for each condition we want

```
data Event:
  | appt(date :: Date, time :: Time,
     duration :: Number, descr :: String)
  | todo(deadline :: Date, task :: String,
     urgency :: String)
end
```

Our List<Event> Data type

Now a calendar can be a **List<Event>**, containing both types of events, e.g.,

```
calendar :: List<Event> =
  [list:
    appt(date(2022, 10, 24), time(10, 30),
      75, "CMPU 101"),
    todo(date(2022, 10, 17),
      “Buy Essential Snacks", "high")]
```

Sherlock Holmes and the *noun* of the missing *plural-noun*…

noun = [list: “case”]
plural-noun = [list: “cases”]

- How do we work with a list where the items can have different parts?
- We’ve already seen the way to work with different varieties of data; it’s **cases**!

Event-matches

- if we want to search our calendar for all events related to a term, we could write a function **event-matches**.
- Let’s go to the pyret IDE.

Event-matches

And we can use it to filter our calendar:

```
fun search-calendar(cal :: List<Event>,
  term :: String) -> List<Event>:
 doc: "Return just the calendar events that contain the term"
 filter(
  lam(e): event-matches(e, term) end,
  cal)
end
```

A word about functions…

The input parameters here are generic

They do not correspond to any existing event list or term!

```
fun search-calendar(cal :: List<Event>,
    term :: String) -> List<Event>:
  doc: "Return just the calendar events that contain the term"
  filter(
    lam(e): event-matches(e, term) end,
    cal)
end
```

Debrief: lists and recursion

- A list is just a built-in kind of conditional data!
- We use **cases** to tell apart its two possibilities – **empty** or **link.**

Debrief: lists and recursion

```
data MyList:
  | my-empty
  | my-link(first, rest :: MyList)
end
```

What's different here?

1. We have a case that's just a special keyword rather than a constructor.
2. Part of the second case" is of the same type we're defining.

Debrief: lists and recursion

```
data MyList:
  | my-empty
  | my-link(first, rest :: MyList)
end
```

What's different here?

1. We have a case that's just a special keyword rather than a constructor.
2. Part of the second case” is of the same type we're defining.
   - A recursive definition!

Debrief: lists and recursion

```
data MyList:
  | my-empty
  | my-link(first, rest :: MyList)
end
```

```
my-empty

my-link(1,
  my-link(2,
    my-link(3,
      my-empty)))
```

What's different here?

1. We have a case that's just a special keyword rather than a constructor.
2. Part of the second case” is of the same type we're defining.
    - A recursive definition!

Using my-list template

And just like we did for a List, we use this template to write a function that recursively processes the data:

```
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
```

Steps to write a generic template

- Given a (recursive) *data definition*, you write a generic template by:
  1. Creating a function header,
  2. Using ***cases*** to break the data input into its variants,
     - In each case, list each of the fields as part of the answer
  3. Calling the function itself on any recursive fields.

Link to code

- 14_new_data_types.arr

Acknowledgements

- This lecture incorporates material from:
- Kathi Fisler, Brown University,
- Marc Smith, Vassar College
- And, Jonathan Gordon, Vassar College