Problem Solving and Abstraction (CMPU 101)

Tom Ellman

Lecture 14
Data Types of Our Own

• PyRet provides several types of data: numbers, strings, images, booleans, tables, and lists.

• These types are broadly useful in many applications.

• But sometimes we need data types of our own.
In Bizzaro World everything is opposite to our world.

- Bizzaro Vassar (BV) needs software to conduct surveillance of Bizarro Vassar students’ (BVS) electronic messages.
- BV *promises* to look only at meta-data and not the contents of BVS’ messages. (Ha!)
- The meta-data includes:
  - Sender
  - Recipient
  - Day of the week
  - Time (hour and minute)
You may want to read this article, which has been censored in Bizarro World.

John Bohannon, "Your call and text records are far more revealing than you think", Science, 2016
We could use a table.

<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>

- What if there are multiple recipients?
- How should we represent time?
  - “4:55” or 455 or [list: 4,55] ...?...
  - Or use two columns (hour, minute) for time.
- Using two columns we can access time components independently.
- Using one column all the time data is in one place.
Let’s define a new data type that has two or more components.

Name of the Data Type

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

Constructor Function that Builds Data of this Type

Components of the Data

Data types with multiple components are sometimes called tuples or records.
After defining the data type:

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

We can call `time` to build `Time` values:
```haskell
>>> noon = time(12, 0)
>>> half-past-three = time(3, 30)
```

We can use dot notation to access the components:
```haskell
>>> noon.hours
12
>>> half-past.mins
30
```
Now our table could 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>
Implement: message-before

- Given:
  - A row representing a message.
  - A Time value: deadline

- Return true if the time of the message is earlier than the deadline. Otherwise return false.
data Time:
  | time(hours :: Number, mins :: Number)
end

messages =
  table:
    sender :: String,
    recipient :: String,
    day :: String,
    time :: Time
    row:  "4015551234", "8025551234", "Mon", time(4, 55)
end
fun message-before(msg :: Row, tm :: Time) -> Boolean:
    doc: "Return true if msg was sent before tm."
    earlier(msg["time"], tm)

where:
    message-before(messages.row-n(0), time(5, 00)) is true
    message-before(messages.row-n(0), time(2, 00)) is false
end

fun earlier(tm1 :: Time, tm2 :: Time) -> Boolean:
    doc: "Return true if time tm1 is before tm2."
    (tm1.hours < tm2.hours)
    or
    ((tm1.hours == tm2.hours) and (tm1.mins < tm2.mins))

where:
    earlier(time(0, 0), time(0, 1)) is true
    earlier(time(0, 1), time(1, 0)) is true
    earlier(time(1, 3), time(1, 2)) is false
    earlier(time(1, 0), time(0, 3)) is false
end
Appointment Calendar

• A calendar is a collection of appointments.
• An appointment has four parts:
  – Date
  – Start Time
  – Duration
  – Description
One Possible Design

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> = ...
Let’s also put tasks on the calendar.

A task has three parts:

– Task
– Deadline
– Urgency
An Event is an **appt** or a **todo**

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

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

calendar :: List<Event> = ...
Now a calendar can contain both types of events.

calendar :: List<Event> =
  [list:
    appt(date(2021, 10, 25), time(13, 30), 75, "CMPU 101"),
    todo(date(2021, 10, 27), "Use avocado", "high")
  ]
search-calendar

• Given:
  – cal :: List<Event>
  – term :: String

• Return a list of all the events on cal for which event-matches(event,term) is true.
event-matches

• Given
  – event :: Event
  – term :: String

• Return true if term appears in either the descr component (of appt) or the task component (of todo). Otherwise return false.
fun event-matches(event :: Event, term :: String) -> Boolean:
   cases (Event) event:
      | appt(d, t, dur, desc) => string-contains(desc, term)
      | todo(dl, task, urg) => string-contains(task, term)
   end
where:
   event-matches(
      appt(date(2021, 10, 25), time(5, 0), 50,
            "Cooking avocados"), "avocado") is true
   event-matches(
      appt(date(2021, 10, 25), time(8, 10), 180,
            "Baseball game"), "avocado") is false
   event-matches(
      todo(date(2021, 10, 25),
            "Use avocado", "high"),
            "avocado") is true
end

Notice that we use a **cases** expression to separately handle appointments (appt) and tasks (todo).
fun search-calendar(cal :: List<Event>, term :: String) -> List<Event>:
    L.filter(lam(e): event-matches(e, term) end, cal)
end
Defining Recursive Data

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

my-list = my-link(1, my-link(2, my-link(3, my-empty)))

# [my-list: 1, 2, 3]

Here we see how we could have defined the list data type ourselves.
fun my-list-length(ml :: MyList) -> Number:
  doc: "Returns length of ml."
  cases (MyList) ml:
    | my-empty => 0
    | my-link(f, r) => 1 + my-list-length(r)
  end
where:
  my-list-length(my-empty) is 0
  my-list-length(my-list) is 3
end

Here we use a cases expression with pattern matching to implement a function on my-list.