Here’s a JSON structure for a newspaper

```json
{
    "authors": [
        {
            "name": "Alice Action",
            "email": "alice@rice.edu"
        },
        {
            "name": "Bob Bigbooté",
            "email": "bob@rice.edu"
        },
        {
            "name": "Charlie Chancery",
            "email": "charlie@rice.edu"
        },
        {
            "name": "Dorothy Danger",
            "email": "dorothy@rice.edu"
        },
        {
            "name": "Eve Ebullience",
            "email": "eve@rice.edu"
        }
    ],
    "articles": [
        {
            "title": "Rice Owls Win Superbowl, Surprised NFL Fans React",
            "authors": ["alice@rice.edu", "bob@rice.edu"],
            "body": "In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34-7 in the Superbowl."
        },
        {
            "title": "Rice Band Deflated Footballs",
            "authors": ["bob@rice.edu"],
            "body": "Rice bandmembers thought they were just making fun of Deflategate, but their prank somehow propelled the Owls to victory."
        },
        {
            "title": "North Servery Receives Three Michelin Stars",
            "authors": ["eve@rice.edu"],
```
Here's an XML version

<newspaper>
  <authors>
    <author>
      <name>Alice Action</name>
      <email>alice@rice.edu</email>
    </author>
    <author>
      <name>Bob Bigbooté</name>
      <email>bob@rice.edu</email>
    </author>
    <author>
      <name>Charlie Chancery</name>
      <email>charlie@rice.edu</email>
    </author>
    <author>
      <name>Dorothy Danger</name>
      <email>dorothy@rice.edu</email>
    </author>
    <author>
      <name>Eve Ebullience</name>
      <email>eve@rice.edu</email>
    </author>
  </authors>
  <articles>
    <article>
      <title>Rice Owls Win Superbowl, Surprised NFL Fans React</title>
      <authors>
        <author>alice@rice.edu</author>
        <author>bob@rice.edu</author>
      </authors>
      <body>
        In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34–7 in the Superbowl.
      </body>
    </article>
    <article>
      <title>Rice Band Deflated Footballs</title>
      <authors>
      </authors>
    </article>
  </articles>
</newspaper>
Are these really any different?

```json
{
    "name": "Alice Action",
    "email": "alice@rice.edu"
},
{
    "name": "Bob Bigbooté",
    "email": "bob@rice.edu"
}
```

```xml
<author>
    <name>Alice Action</name>
    <email>alice@rice.edu</email>
</author>
<author>
    <name>Bob Bigbooté</name>
    <email>bob@rice.edu</email>
</author>
```
Are these really any different?

Different syntax
XML angle brackets vs. JSON curly brackets, etc.

Mostly the same meaning
XML forces each “entity” to have a declared type (“author”, “name”, etc.)
JSON allows name declarations (as below) but doesn’t require it (e.g., in arrays)

```json
{
    "name": "Alice Action",
    "email": "alice@rice.edu"
},
{
    "name": "Bob Bigbooté",
    "email": "bob@rice.edu"
}
```

```xml
<author>
    <name>Alice Action</name>
    <email>alice@rice.edu</email>
</author>
<author>
    <name>Bob Bigbooté</name>
    <email>bob@rice.edu</email>
</author>
```
Syntax vs. semantics

We generally want to require something beyond **syntactic** correctness
Not just that it’s well-formed JSON or well-formed XML

**Examples**
Every author has a name and an email
Every article has a non-empty list of authors, a title, a section, and a body

```json
{
    "name": "Alice Action",
    "email": "alice@rice.edu"
},
{
    "name": "Bob Bigbooté",
    "email": "bob@rice.edu"
}
```
The Java compiler worries about this

Every variable or method used has been declared beforehand
Look around to make sure you’ve seen it

The types all match up
You can’t just say something like `String x = 34 + 19;`
This gets more complicated with generics, lambdas, and type inference

Take Comp412 and learn all about static semantics analysis
But we have to deal with a simpler version of the problem now
In next week’s text adventure game...

You’ll have some basic concepts like Room, Item, Weapon, etc. And you’ll want to read them in from a JSON file.

But how do you do that?
Option 1: just read the JSON data in and start working directly with it
Option 2: convert from JSON to something more specific to your design
Let’s go back to our newspaper example

Just like with our JSON “builders”, we want to have public builders and private constructors
You’ll see all this code on Monday in edu.rice.week9newspaper:

```java
public class DB {
    public static class Author {
        public final String email;
        public final String name;
        private Author(String email, String name) {
            this.email = email;
            this.name = name;
        }
    }
}
```

So far, it’s all pretty simple, right?
Cute helper functions

It's useful to be able to go back to JSON (for printing, for saving, etc.)

```java
public Parser.IValue toIValue() {
    return jobject(jpair("email", email), jpair("name", name));
}

public String toString() {
    return toIValue().toString();
}
```
public static Optional<Author> build(Parser.IValue input) {
    // we require a name and an email field
    Optional<String> oname = Utils.o getPath(input, "name")
        .flatMap(Parser.IValue::asOJString)
        .map(Parser.JString::toUnescapedString);

    Optional<String> oemail = Utils.o getPath(input, "email")
        .flatMap(Parser.IValue::asOJString)
        .map(Parser.JString::toUnescapedString);

    // we require an author to have a name and an email
    if (oname.isPresent() && oemail.isPresent())
        return Optional.of(new Author(oemail.get(), oname.get()));
    else {
        Log.e(TAG, () -> "failed to read valid author: " + input.toString());
        return Optional.empty();
    }
}
More Optional!

Optional<String> oname = Utils.ogetPath(input, "name")

.flatMap(Parser.IValue::asOJString)

.map(Parser.JString::toUnescapedString);
Optional<String> oname = Utils.ogetPath(input, "name")

New! edu.rice.json.Utils: query a “path” into the JSON structure, returns Optional<IValue>

  .flatMap(Parser.IValue::asOJString)

  .map(Parser.JString::toUnescapedString);
Optional<String> oname = Utils.ogetPath(input, "name")

flatMap(Parser.IValue::asOJString)

New! IValue.asOJString: if it’s a JString, return Optional<JString>, otherwise Optional.empty()

.map(Parser.JString::toUnescapedString);
Optional<String> oname = Utils.o getPath(input, "name")

.flatMap(Parser.IValue::asOJString)

.map(Parser.JString::toUnescapedString);

Convert to a “normal” Java String, if it’s there, otherwise more Optional.empty()
JSON Utilities: ogetPath

Lets you look deeply into a JSON data structure

```java
public static Optional<Parser.IValue> ogetPath(Parser.IValue iValue, String path)
```

Path is like a Unix file path ("a/b/c")
ogetPath in action

```json
{
    "authors": [
        {
            "name": "Alice Action",
            "email": "alice@rice.edu"
        },
        {
            "name": "Bob Bigbooté",
            "email": "bob@rice.edu"
        },
        {
            "name": "Charlie Chancery",
            "email": "charlie@rice.edu"
        },
        {
            "name": "Dorothy Danger",
            "email": "dorothy@rice.edu"
        },
        {
            "name": "Eve Ebullience",
            "email": "eve@rice.edu"
        }
    ],

    "articles": [
        {
            "title": "Rice Owls Win Superbowl, Surprised NFL Fans React",
            "authors": ["alice@rice.edu", "bob@rice.edu"],
            "body": "In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34-7 in the Superbowl."
        },
        {
            "title": "Rice Band Deflated Footballs",
            "authors": ["bob@rice.edu"],
            "body": "Rice bandmembers thought they were just making fun of Deflategate, but their prank somehow propelled the Owls to victory."
        },
        {
            "title": "North Servery Receives Three Michelin Stars",
            "authors": ["eve@rice.edu"],
            "body": "The North Servery has been awarded three Michelin Stars, a testament to their outstanding culinary skills."
        }
    ]
}
```

Utils.ogetPath(rootVal, “articles/0/title”)  
--> Optional.of(“Rice Owls Win Superbowl...”)
Utils.ogetPath(rootVal, "articles/0/title")
--> Optional.of("Rice Owls Win Superbowl...")
o getPath in action

```json
{
  "authors": [
    {
      "name": "Alice Action",
      "email": "alice@rice.edu"
    },
    {
      "name": "Bob Bigbooté",
      "email": "bob@rice.edu"
    },
    {
      "name": "Charlie Chancery",
      "email": "charlie@rice.edu"
    },
    {
      "name": "Dorothy Danger",
      "email": "dorothy@rice.edu"
    },
    {
      "name": "Eve Ebullience",
      "email": "eve@rice.edu"
    }
  ],
  "articles": [
    {
      "title": "Rice Owls Win Superbowl, Surprised NFL Fans React",
      "authors": ["alice@rice.edu", "bob@rice.edu"],
      "body": "In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34-7 in the Superbowl."
    },
    {
      "title": "Rice Band Deflated Footballs",
      "authors": ["bob@rice.edu"],
      "body": "Rice bandmembers thought they were just making fun of Deflategate, but their prank somehow propelled the Owls to victory."
    },
    {
      "title": "North Servery Receives Three Michelin Stars",
      "authors": ["eve@rice.edu"]
    }
  ]
}
```

Utils.o getPath(rootVal, “articles/0/title”) --> Optional.of(“Rice Owls Win Superbowl...”)
ogetPath in action

```java
Utils.ogetPath(rootVal, "articles/0/title")
--> Optional.of("Rice Owls Win Superbowl...")
```

```javascript
{
    "authors": [
        {
            "name": "Alice Action",
            "email": "alice@rice.edu"
        },
        {
            "name": "Bob Bigbooté",
            "email": "bob@rice.edu"
        },
        {
            "name": "Charlie Chancery",
            "email": "charlie@rice.edu"
        },
        {
            "name": "Dorothy Danger",
            "email": "dorothy@rice.edu"
        },
        {
            "name": "Eve Ebullience",
            "email": "eve@rice.edu"
        }
    ],
    "articles": [
        {
            "title": "Rice Owls Win Superbowl, Surprised NFL Fans React",
            "authors": ["alice@rice.edu", "bob@rice.edu"],
            "body": "In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34-7 in the Superbowl."
        },
        {
            "title": "Rice Band Deflated Footballs",
            "authors": ["bob@rice.edu"],
            "body": "Rice bandmembers thought they were just making fun of Deflategate, but their prank somehow propelled the Owls to victory."
        },
        {
            "title": "North Servery Receives Three Michelin Stars",
            "authors": ["eve@rice.edu"],
```
o getPath in action

```javascript
Utils.o getPath(rootVal, "articles/0/title")
--> Optional.of("Rice Owls Win Superbowl...")
```

```
{
  "authors": [
    {
      "name": "Alice Action",
      "email": "alice@rice.edu"
    },
    {
      "name": "Bob Bigbooté",
      "email": "bob@rice.edu"
    },
    {
      "name": "Charlie Chancery",
      "email": "charlie@rice.edu"
    },
    {
      "name": "Dorothy Danger",
      "email": "dorothy@rice.edu"
    },
    {
      "name": "Eve Ebullience",
      "email": "eve@rice.edu"
    }
  ],
  "articles": [
    {
      "title": "Rice Owls Win Superbowl, Surprised NFL Fans React",
      "authors": ["alice@rice.edu", "bob@rice.edu"],
      "body": "In a sports upset few could have imagined, the Rice Owls defeated the New England Patriots 34-7 in the Superbowl."
    },
    {
      "title": "Rice Band Deflated Footballs",
      "authors": ["bob@rice.edu"],
      "body": "Rice bandmembers thought they were just making fun of Deflategate, but their prank somehow propelled the Owls to victory."
    },
    {
      "title": "North Servery Receives Three Michelin Stars",
      "authors": ["eve@rice.edu"],
      "body": "North Servery received three Michelin stars, the highest accolade in the culinary world, for its exceptional cuisine and impeccable service."
    }
  ]
}
```
Also, regular expression querying!

Given a list of regexs, returns a list of IVValues with matching paths

```csharp
public static IList<Parser.IValue> getPathMatchesRegex(
    Parser.IValue iValue,
    IList<String> pathRegexList)
```

This is fun stuff! Full-blown regex matching on keys!
Array indices are numbers. First converted to strings, then pass to the regex.

Why? So it can be regex’s all the way down.
- The regex strings are internally converted to Predicate functions, which you can alternately use if you want to be super-duper general-purpose.
getPathMatchesRegex in action

```java
Utils.getPathMatchesRegex(rootVal, List.of(".*", "name"))
--> List.of("Alice Action", "Bob Bigbooté", ...)
```
You can also update a JSON expression

Still functional programming here, so you get out a new one

```java
public static Optional<Parser.IValue> updatePath(
    Parser.IValue iValue,
    String path,
    Function<Optional<Parser.IValue>, Optional<Parser.IValue>> updateF)
```

If there’s a value at that path, it’s passed to the function
Otherwise Optional.empty()

If the return value is Optional.of(something), that’s the new value
If the return value is Optional.empty(), the key/value are removed!

If the entire path isn’t there at all, but the function returns Optional.of(…)
Then the entire path is created!

There’s even a regex version that will update all matching values
You can also update a JSON expression

Still functional programming here, so you get out a new one

```java
public static Optional<Parser.IValue> updatePath(
    Parser.IValue iValue,
    String path,
    Function<Optional<Parser.IValue>, Optional<Parser.IValue>> updateF)
```

If there’s a value at that path, it’s passed to the function.
Otherwise Optional.empty()

If the return value is Optional.of(something), that’s the new value.
If the return value is Optional.empty(), the key/value are removed!

If the entire path isn’t there at all, but the function returns Optional.of(...) Then the entire path is created!

There’s even a regex version that will update all matching values

Optional is amazingly useful
So what should you use in your game?

It’s mostly up to you, but...

You should use JSON for your input format

*Top tip:* make a file with IntelliJ, call it “whatever.json”, and IntelliJ will help you!

   Reading from a file, rather than just a String in memory is annoying, but not too bad

You should imagine actions in the game as functions that update your state

*Top tip:* printing a JSON structure will help you debug your system state

Why functional again?

You can test individual modules against individual inputs

Unit tests will be required, best when they can operate without the web front-end
Digression: Non-human-readable data

Human-readable data (JSON, XML, etc.) is hugely useful
Just pull it up in a text editor, print it in a log, etc.

It’s also really expensive to parse and load into your data structures
Completely unacceptable for “scientific” workloads
Wasteful in bandwidth-limited environments

The solution? We need something that’s...
Portable: works across different programming languages, different computers
Compact/fast: quick to read, quick to write, efficient use of bandwidth
Extensible: lets you add new fields without breaking old code
Two popular solutions: Protobufs & Thrift

Protocol buffers: invented at Google, widely supported
Thrift: invented at Facebook, widely supported

Non-solutions:

Java serialization: surprisingly slow, also a giant security hole

ASN.1: mostly of historical interest, invented by telephony people, hugely complex, and sometimes also a giant security hole
  Yet still part of the SSL/TLS standard, used every time you connect to an encrypted web site

JSON or XML, then gzip: uses less space, but uses lots more CPU
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
Protobufs, in a nutshell

package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}

enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
Protobuf “compiler” writes code for you
Protobuf “compiler” writes code for you

Reads the “.proto” file, writes “.java” files (or tons of other languages)

Simple datatypes with simple constructors
Producing protobufs is no more complicated than toString()
Protobuf "compiler" writes code for you

Reads the " .proto " file, writes " .java " files (or tons of other languages)
Simple datatypes with simple constructors
Producing protobufs is no more complicated than toString()

YOU’RE NOT REQUIRED TO DO THIS IN COMP215
But it’s really cool to know there are good alternatives beyond JSON
Protobuf “compiler” writes code for you

Reads the “.proto” file, writes “.java” files (or tons of other languages)
Simple datatypes with simple constructors
Producing protobufs is no more complicated than toString()

YOU’RE NOT REQUIRED TO DO THIS IN COMP215
But it’s really cool to know there are good alternatives beyond JSON

And, oh by the way...
Protobuf “compiler” writes code for you

Reads the “.proto” file, writes “.java” files (or tons of other languages)
Simple datatypes with simple constructors
Producing protobufs is no more complicated than `toString()`

YOU’RE NOT REQUIRED TO DO THIS IN COMP215
But it’s really cool to know there are good alternatives beyond JSON

And, oh by the way...

They don’t support mutation, either.
This has all been done before...

The HTML/XML universe had tons of tools to read/write/update

Standard specs (XPath & XQuery) make it easy to dig down
Nice list of XPath query examples:

Also relevant: XSD (“XML Schema Definitions”)
Specify structural requirements (e.g., every author must have a name)

And many open-source libraries that can do similar things for JSON
JSONiq, JSONPath, JSPath, JSONSelect, and more.
But... you don’t need any of this stuff for Comp215.
Live coding

Guided tour of the newest features in edu.rice.json