COMP60411: Modelling Data on the Web
Tree Data Models
Week 2

Tim Morris & Uli Sattler
University of Manchester
Reminder: Plagiarism & Academic Malpractice

• We assume that you have all by now successfully completed the **Plagiarism and Malpractice Test**

• ...if you haven’t: do so **before** you submit **any** coursework (assignment or assessment)

• ...because we work under the assumption that
  – you know what you do
  – you take pride in your own thoughts & your own writing
  – you don’t steal thoughts or words from others

• ...and if you don’t, and submit coursework where you have **copied other people’s work without correct attribution**
  it costs you **at least** marks or more, e.g., your MSc
Reminder

We maintain 3 sources of information:

• **syllabus** …/pgt/COMP60411/syllabus/
• **materials** …/pgt/COMP60411/
  - growing continuously
  - with slides, reading material, etc
  - with TA lab times
• **Blackboard via myManchester**
  - growing continuously
  - Forums
    • General
    • Week 1, Week 2, …
  - Coursework
Coursework - Week 1

• Q1: looks good, will look better next week, BUT…
• SE1: looks mostly good
  • use a good spell & grammar checker!
  • answer the question!
    • We know what ER diagrams are, no need to explain them
    • No need to explain logical/physical model in detail!
  • avoid non sequiturs
• M1:
  • …
• CW1:
  • …

• For all:
  • check our feedback in the rubrics
  • if you can’t find them, ask us in labs
  • start in time
Today

We will encounter many things:

**Tree data models:**

1. Data Structure formalisms: JSON
2. Schema Language: JSON Schema
3. Data Manipulation: Python, JSON package

**General concepts:**

- Semi-structured data
- Self-Describing
- Trees
- Regular Expressions
- Internal & External Representation, Parsing, Serialisation
- Validation, valid, …
- Format
Extending Last Week’s Running Example
Extended Running Example

• Remember last week’s example:
  – per person 1-3 data records, with address, phone, email,…
• now combine this with **management information**:
  – who supervises/line manages whom?

  **Employees**

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Postcode</th>
<th>City</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234123</td>
<td>M16 0P2</td>
<td>Manchester</td>
<td>…</td>
</tr>
<tr>
<td>1234124</td>
<td>M2 3OZ</td>
<td>Manchester</td>
<td>…</td>
</tr>
<tr>
<td>1234567</td>
<td>SW1 A</td>
<td>London</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

  **Management**

<table>
<thead>
<tr>
<th>Manager ID</th>
<th>Managee ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234124</td>
<td>1234123</td>
</tr>
<tr>
<td>1234567</td>
<td>1234124</td>
</tr>
<tr>
<td>1234124</td>
<td>1234567</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

• …what could go wrong?
• …what did go wrong?
• Take a few minutes and sketch this SQL query:

Q1: all postcodes of 4th-level managers

### Employees

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Postcode</th>
<th>City</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1234123</td>
<td>M16 0P2</td>
<td>Manchester</td>
<td></td>
</tr>
<tr>
<td>1234124</td>
<td>M2 3OZ</td>
<td>Manchester</td>
<td></td>
</tr>
<tr>
<td>1234567</td>
<td>SW1 A</td>
<td>London</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Management

<table>
<thead>
<tr>
<th>Manager ID</th>
<th>ManageeID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234124</td>
<td>1234123</td>
</tr>
<tr>
<td>1234567</td>
<td>1234124</td>
</tr>
<tr>
<td>1234123</td>
<td>1234567</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Q1: Tricky..

Q1’: Postcodes of all managers:

```
SELECT Postcode
FROM Employees E, Management M
WHERE E.EmployeeID = M.ManagerID
```

Q1”: Postcode of 2nd level managers:

```
SELECT Postcode
FROM Employees E
INNER JOIN
(SELECT ManagerID
 FROM Management M1, Management M2
 WHERE M1.ManageeID = M2.ManagerID) M
ON E.EmployeeID = M.ManagerID
```

…more and more joins!
Running Example (2)

- Take a few minutes and sketch this SQL query:
  Q2: “error” if we have a **cyclic** management structure

<table>
<thead>
<tr>
<th>Employees</th>
<th>Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee ID</td>
<td>Postcode</td>
</tr>
<tr>
<td>1234123</td>
<td>M16 0P2</td>
</tr>
<tr>
<td>1234124</td>
<td>M2 3OZ</td>
</tr>
<tr>
<td>1234567</td>
<td>SW1 A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Q2: Tricky…

– Detecting management cycles of length 1:

![Management Table]

– Detecting management cycles of length 2:

```
SELECT EmployeeID
FROM Management M
WHERE M.ManageeID = M.ManagerID
```

```
SELECT EmployeeID
FROM Employees E1
INNER JOIN
(SELECT EmployeeID
 FROM Management M1, Management M2
 WHERE M1.ManageeID = M2.ManagerID) M
ON E1.EmployeeID = M.ManagerID
```

– …where do we stop?
A new example: UniProt, a Protein Database

• A research community based & curated knowledge base of
  – 550K protein sequences,
  – comprising 192M amino acids
  – abstracted from 220K references.
• Proteins largely determine how (parts of) living things work and interact
  – how/where diseases work
• Used for a variety of research into
  – (causes of) diseases
  – genetics
  – (personalized) drugs
  – …
**Protein** | Fanconi anemia group J protein homolog
---|---
**Gene** | Brip1
**Organism** | Mus musculus (Mouse)
**Status** | Reviewed - Annotation score: 5 - Experimental evidence at transcript level

### Function

DNA-dependent ATPase and 5' to 3' DNA helicase required for the maintenance of chromosomal stability. Acts late in the 'Fanconi anemia' pathway, after FANCD2 ubiquitination. Involved in the repair of DNA double-strand breaks by homologous recombination in a manner that depends on its association with BRCA1 (By similarity).

**Catalytic activity**

ATP + H₂O → ADP + phosphate.

**Cofactor**

[4Fe-4S] cluster  [By similarity]

**Note:** Binds 1 [4Fe-4S] cluster.  [By similarity]

### Sites

<table>
<thead>
<tr>
<th>Feature key</th>
<th>Position(s)</th>
<th>Description</th>
<th>Actions</th>
<th>Graphical view</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal binding</td>
<td>286</td>
<td>Iron-sulfur (4Fe-4S)  [By similarity]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metal binding</td>
<td>301</td>
<td>Iron-sulfur (4Fe-4S)  [By similarity]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metal binding</td>
<td>313</td>
<td>Iron-sulfur (4Fe-4S)  [By similarity]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metal binding</td>
<td>353</td>
<td>Iron-sulfur (4Fe-4S)  [By similarity]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Protein data from UniProt

UniProt

• provides a **web query interface** to Uniprot DB,
  – manual
  – programmatic
• e.g., query [http://www.uniprot.org/uniprot/](http://www.uniprot.org/uniprot/) for ‘BRCA’

• ...biologists need to integrate, share, query, analyse, and search this data

• ...so what format is/should it be in?
• ...or what format should it be made available in to be integrated with other data?
Protein data from UniProt in as text

<table>
<thead>
<tr>
<th>ID</th>
<th>FANCJ_MOUSE</th>
<th>Reviewed; 1174 AA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Q5SXJ3; Q8BJQ8; Q8BKI6;</td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td>25-OCT-2005, integrated into UniProtKB/Swiss-Prot.</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>RecName: Full=Fanconi anemia group J protein homolog;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>Short=Protein FACJ;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>EC=3.6.4.13;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>AltName: Full=ATP-dependent RNA helicase BRIP1;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>AltName: Full=BRCA1-associated C-terminal helicase 1;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>AltName: Full=BRCA1-interacting protein C-terminal helicase 1;</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>Short=BRCA1-interacting protein 1;</td>
<td></td>
</tr>
<tr>
<td>GN</td>
<td>Name=Bripl; Synonyms=Bach1, Fancj;</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Mus musculus (Mouse).</td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;</td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia; Myomorpha;</td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>Muroidea; Muridae; Murinae; Mus; Mus.</td>
<td></td>
</tr>
<tr>
<td>OX</td>
<td>NCBI_TaxID=10090;</td>
<td></td>
</tr>
<tr>
<td>RN</td>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>NUCLEOTIDE SEQUENCE [LARGE SCALE GENOMIC DNA].</td>
<td></td>
</tr>
<tr>
<td>RC</td>
<td>STRAIN=C57BL/6J;</td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td>PubMed=19468303; DOI=10.1371/journal.pbio.1000112;</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Church D.M., Goodstadt L., Hillier L.W., Zody M.C., Goldstein S.,</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>She X., Bult C.J., Agarwala R., Cherry J.L., DiCuccio M., Hlavina W.,</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Kapustin Y., Meric P., Maglott D., Birtle Z., Marques A.C., Graves T.,</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Zhou S., Teague B., Potamousis K., Churas C., Place M., Herschleb J.,</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Runnheim R., Forrest D., Amos-Landgraf J., Schwartz D.C., Cheng Z.,</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Lindblad-Toh K., Eichler E.E., Ponting C.P.;</td>
<td></td>
</tr>
<tr>
<td>RT</td>
<td>&quot;Lineage-specific biology revealed by a finished genome assembly of the mouse.&quot;;</td>
<td></td>
</tr>
</tbody>
</table>
# Protein data from UniProt in a table (1)

| Protein Full Name                  | Short Name | Alternative Name 1 | Alternative Name 2 | Alternative Name 3 | Gene 1 | Gene 2 | Gene 3 | ... | Organism     | Taxon 1      | Taxon 2           | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fanconi anemia group J</td>
<td>FACJ</td>
<td>ATP-dependent RNA helicase</td>
<td>BRCA 1-interacting protein C-termin</td>
<td>BRCA 1-interacting protein 1</td>
<td>BRIP1</td>
<td>BACH 1</td>
<td>FANCJ</td>
<td></td>
<td>Halorhodospirillum HF2</td>
<td>Viruses</td>
<td>dsDNA viruses, no RNA stage</td>
<td>...</td>
</tr>
<tr>
<td>ATP-dependent helicase</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td>Gallus gallus / Chicken</td>
<td>Eukaryota</td>
<td>Metazoa</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>


Protein data from UniProt in many tables (2)

Proteins

<table>
<thead>
<tr>
<th>Protein ID</th>
<th>Full Name</th>
<th>Short Name</th>
<th>Organism</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234123</td>
<td>Fanconi anemia</td>
<td>FACJ</td>
<td>Halorubrum phage HF2</td>
<td>...</td>
</tr>
<tr>
<td>1234567</td>
<td>ATP-dependent</td>
<td>N/A</td>
<td>Gallus gallus / Chicken</td>
<td>...</td>
</tr>
</tbody>
</table>

Protein-names

<table>
<thead>
<tr>
<th>Protein ID</th>
<th>Alternative Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234123</td>
<td>ATP-dependent RNA helicase BRIP1</td>
</tr>
<tr>
<td>1234123</td>
<td>BRCA1-interacting protein C-terminal helicase 1</td>
</tr>
<tr>
<td>1234123</td>
<td>BRCA1-interacting protein 1</td>
</tr>
</tbody>
</table>

Protein-genes

<table>
<thead>
<tr>
<th>Protein ID</th>
<th>Genes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234123</td>
<td>BRIP1</td>
</tr>
<tr>
<td>1234123</td>
<td>BACH1</td>
</tr>
<tr>
<td>1234567</td>
<td>helicas</td>
</tr>
</tbody>
</table>

... too many joins!
Protein data from UniProt in JSON

```json
[{
  "accession": "Q9NXB0-3",
  "id": "MKS1-3_HUMAN",
  "proteinExistence": "Evidence at protein level",
  "info": {
    "type": "Swiss-Prot",
    "created": "2006-03-07",
    "modified": "2019-09-18",
    "version": 135
  },
  "organism": {
    "taxonomy": 9606,
    "names": [{
      "type": "scientific",
      "value": "Homo sapiens"
    }, {
      "type": "common",
      "value": "Human"
    }],
}
]"}
Protein data from UniProt in an XML doc (1)

<?xml version="1.0" encoding="UTF-8"?>
  xsi:schemaLocation="http://uniprot.org/uniprot http://www.uniprot.org/support/docs/uniprot.xsd">
  <entry dataset="Swiss-Prot" created="2005-01-04" modified="2010-08-10" version="80">
    <accession>Q9BX63</accession>
    <accession>Q3MJE2</accession>
    <accession>Q8NCI5</accession>
    <name>FANCJ_HUMAN</name>
    <protein>
      <recommendedName ref="1">
        <fullName>Fanconi anemia group J protein</fullName>
        <shortName>Protein FACJ</shortName>
      </recommendedName>
      <alternativeName>
        <fullName>ATP-dependent RNA helicase BRIP1</fullName>
      </alternativeName>
      <alternativeName>
        <fullName>BRCA1-interacting protein C-terminal helicase 1</fullName>
      </alternativeName>
      <alternativeName>
        <fullName>BRCA1-associated C-terminal helicase 1</fullName>
      </alternativeName>
    </protein>
    <gene>
      <name type="primary">BRIP1</name>
      <name type="synonym">BACH1</name>
      <name type="synonym">FANCJ</name>
    </gene>
  </entry>
</uniprot>
Protein data from UniProt in an XML doc (2)

<organism>
  <name type="scientific">Homo sapiens</name>
  <name type="common">Human</name>
  <dbReference type="NCBI Taxonomy" id="9606" key="2"/>
  <lineage>
    <taxon>Eukaryota</taxon>
    <taxon>Metazoa</taxon>
    <taxon>Chordata</taxon>
    <taxon>Craniata</taxon>
    <taxon>Vertebrata</taxon>
    <taxon>Euteleostomi</taxon>
    <taxon>Mammalia</taxon>
    <taxon>Eutheria</taxon>
    <taxon>Euarchontoglires</taxon>
    <taxon>Primates</taxon>
    <taxon>Haplorrhini</taxon>
    <taxon>Catarrhini</taxon>
    <taxon>Hominidae</taxon>
    <taxon>Homo</taxon>
  </lineage>
</organism>

<reference key="3">
  <citation type="journal article" date="2001" name="Cell" volume="105" first="149" last="160">
    <title>BACH1, a novel helicase-like protein, interacts directly with BRCA1 and contributes to its DNA repair function.</title>
    <authorList>
      <person name="Cantor S.B."/>
      <person name="Bell D.W."/>
      <person name="Ganesan S."/>
      <person name="Kass E.M."/>
      <person name="Drapkin R."/>
    </authorList>
  </citation>
</reference>
Another example: arithmetic expressions

- Consider an eLearning site like
  - https://www.kidzone.ws/math/quiz.html or
  - https://math-quiz.co.uk/
Another example: arithmetic expressions

- How to store arithmetic expressions involving operands
  - plus, minus, times, divided-by, …?
- so that parsing and computing values is easy?
  - so that we can easily build a revision/practice web site
  - with many auto-generated, auto-graded questions
  - including parentheses, subexpressions, precedence, nesting…
  - so that parsing/evaluation is easy?
    - Restricting to ‘shallow level’? Noooo!

- String? Table?
- Tree! Aka parse treee:
  - “natural”: 56+42:14
Two pain points common to both examples

Storing data in RDBMs/tables may require

- **Many joins**
  - due to irregular structure
    - varying number of ‘values’ for certain attributes
    - e.g., phone number, email, …
    - e.g., author, alternative name, Protein Names
      - making queries tricky/complicated, thus easy-to-get-wrong
  
- **Recursive joins**
  - due to unbounded depth, e.g., of
    - management tree & cycle management
    - arithmetic expressions
Alternative to Tables: Semi-Structured Data Models
Database Alternatives to Tables

- **Trees**, underlying various **semi-structured data models**:
  - OEM
  - Lore
  - JSON
  - XML

- **Graphs**

  - what are they?
  - what are they good at?
  - Schema Languages: how do we describe ‘legal structures’?
  - Data Manipulation: how do we interact with them?
The Basics First: Semi-structured data

- predates XML
- is an attempt to reconcile
  - (Web) document view and
  - (DB) strict structures
- is data organised in semantic entities, where
  - similar entities are grouped together
  - entities in same group may not have same fields
- often defined as a **possibly nested set** of field-value pairs
- order of fields is not necessarily important
  - e.g.: do we have sets or lists of telephone numbers?
  - ….. fixing an order allows to give meaning to rank
- not all fields may be required
- carries its own description

```
{name: {first:”Uli”, last: “Sattler”},
tel: 56176,
email:”sattler@cs.man.ac.uk”}
```

there is structure!

but not too much structure!

aka **attribute-value** pairs
The Basics First: Semi-structured data

**Example (ctd):**
Values can in turn be **structured**:

```
{name: {first:"Uli", last: “Sattler"},
tel: 56176,
email:"sattler@cs.man.ac.uk"}
```

And we can have **several values** for the same field:

```
{name: {first:"Uli", last: “Sattler"},
tel: 56176,
tel: 56182,
email:"sattler@cs.man.ac.uk"}
```
The Basics First: Semi-structured data

**Important**: are field-value pairs **lists** or **sets**?

I.e., is

```
{name: {first:"Uli", last: “Sattler"},
tel: 12345,
tel: 56176,
email:”sattler@cs.man.ac.uk”}
```

the same as

```
{name: {first:"Uli", last: “Sattler"},
tel: 56176,
tel: 12345,
email:”sattler@cs.man.ac.uk”}
```

(yes if f-v-ps are **sets**, no if they are **lists**)
The Basics First: Semi-structured data

**Important:** does white space matter?

I.e., is

```
{name: {first:”Uli”, last: “Sattler"},
 tel: 56182,
 tel: 56176,
 email:”sattler@cs.man.ac.uk”}
```

the same as

```
{name: {first:”Uli”, last: “Sattler"},
 tel: 56182 ,
 tel: 56176,
 email:”sattler@cs.man.ac.uk”}
```
We need an Internal Representation to know when two pieces of semi-structured data are the same, and to determine what matters.
## External & Internal Representation

<table>
<thead>
<tr>
<th>Level</th>
<th>Data Unit Example</th>
<th>Information Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree/object/…</td>
<td><img src="image" alt="Tree diagram" /></td>
<td>nesting, matching parentheses</td>
</tr>
<tr>
<td>Token</td>
<td><code>_&quot;name&quot;</code>: <code>{&quot;first&quot;:&quot;Bob&quot;, _</code></td>
<td>separator, types,…</td>
</tr>
<tr>
<td>Character</td>
<td><code>_&quot;name&quot;</code>: <code>{&quot;first&quot;:&quot;Bob&quot;, _</code></td>
<td>character encoding</td>
</tr>
<tr>
<td>Bit</td>
<td>10011010</td>
<td></td>
</tr>
</tbody>
</table>

- **Tree/object/…**
  - Example of a tree data structure with nodes for `name`, `first`, and `last`.
  - Matching parentheses are used to indicate nesting.

- **Token**
  - JSON example: `{"first":"Bob", `_`}

- **Character**
  - JSON example: `{"first":"Bob", `_`}

- **Bit**
  - Bit representation: `10011010`
The Basics First: trees as InternRepr for SSD

Let's view/treat nested field-value pairs as trees

```
{name: {first: "Uli", last: "Sattler"},
tel: 56176,
tel: 56182,
email: "sattler@cs.man.ac.uk"}
```
The Basics First: trees as InternRepr for SSD

Let's view/treat nested field-value pairs as trees

Is this the same or a different tree?
Is this the same or different data?
The Basics First: trees as InternRepr for SSD

- In general, a piece of SSD/nested set of field-value pairs, can be represented as a tree
  - leaf nodes standing for single data items
  - inner nodes carry no label
  - edges labelled with field names

```
{name: {first:"Uli", last: "Sattler"},
tel: 56182,
tel: 56176,
email:”sattler@cs.man.ac.uk”}
```
Semi-structured data: tuples with variations

We can easily represent **nested tuples**

```
[[[Uli, Sattler], 56176, sattler@cs.man.ac.uk],
[Bijan, 56183, 783 4672, bparsia@cs.man.ac.uk],
[Leo, 8488342, leo@gmx.com]]
```

as sets of field-value pairs

even if they have *missing* or *duplicated* pairs

...best if we know which element belongs to what

e.g., is “783 4672” Bijan’s telephone number? his email address? age?

```
{person:
    {name: {first: “Uli”, last: “sattler}, tel: 56176, email: “sattler@cs.man.ac.uk”}
person:
    {name: “Bijan”, tel: 56183, tel: 783 4672, email: “bparsia@cs.man.ac.uk”}
person:
    {name: “Leo”, tel: 8488342, email: “leo@gmx.com”})
```
Semi-structured data: tuples with variations

We can easily represent nested tuples

\[
\text{[[[Uli, Sattler], 56176, sattler@cs.man.ac.uk],} \\
\text{[Bijan, 56183, 783 4672, bparsia@cs.man.ac.uk],} \\
\text{[Leo, 8488342, leo@gmx.com]]}
\]

as sets of field-value pairs
   even if they have missing or duplicated pairs
   ...but also without knowing role of elements:

\[
\{1: \\
\quad \{1: \text{“Uli”, 2: “sattler”, 2: 56176, 3: “sattler@cs.man.ac.uk”}\}\}
\]

\[
\begin{align*}
2: \\
\quad \{1: \text{“Bijan”, 2: 56183, 3: 783 4672, 4: “bparsia@cs.man.ac.uk”}\}
\end{align*}
\]

\[
\begin{align*}
3: \\
\quad \{1: \text{“Leo”, 2: 8488342, 3: “leo@gmx.com”}\}
\end{align*}
\]
SSD: representing relational data

Consider two relations:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td></td>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>S</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c2</td>
<td>d2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c3</td>
<td>d3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c4</td>
<td>d4</td>
<td></td>
</tr>
</tbody>
</table>

and their tree representation:

we can represent relational data, though with an overhead
SSD: representing object databases

• we can represent data from object-oriented DBMSs or SE as SSD
  – provided we have object identifiers, e.g., &o1
  – so that objects can refer to each other

Example: 
```json
{ persons: 
  {person: &o1 {
    name: “John”,
    age: 47,
    relatives: {child: &o2,
                child: &o3}}
  person: &o2 {
    name: “Mary”,
    age: 21,
    relatives: {father: &o1,
               sister: &o3}}
  person: &o3 {
    name: “Paula”,
    age: 23,
    relatives: {father: &o1,
               sister: &o2}}} }}
```

⇒ Draw a graph representation of this piece of semi-structured data!
SSD: how to represent/store

- There are various formalisms to store semi-structured data
  - For example
    - Object Exchange Model (OEM, close to previous examples)
    - Lore
    - XML
    - JSON
- Different formalisms with different
  - Internal representations
  - Mechanisms for self-describing
  - Datatypes (e.g., integer, Boolean, string, data-time...) supported
  - Description mechanisms for (semi) structure:
    - Schema languages to describe
      - Which fields are allowed/required where
      - Which values allowed/required where
    - Query languages & manipulation mechanisms
JSON

a tree-shaped/semi-structured data model/interchange format

http://www.json.org
JavaScript Object Notation

- JSON was developed to serialise/store/transmit/… JavaScript objects
  - other programming languages can read/write JSON as well
    - incl. Python, Java, …

- JS objects (or other data) can be *serialised* into
  - JSON: basically automatic
  - XML: involves **design choices**
    - attribute or child element?
    - element/attribute names?
JavaScript Object Notation - JSON

- Javascript has a rich set of literals (ext. reps) called **items**
  - **Atomic** (numbers, booleans, strings*)
    - 1, 2, true, “I’m a string”
  - **Composite: Arrays**
    - Ordered lists/vectors of **items** with random access
    - e.g., [1, 2, “one”, “two”]
  - **Composite: “Objects”**
    - Sets/unordered lists/associative arrays/dictionary
    - e.g., {“one”:1, “two”:2} but not {“one”:1, “one”:2}
    - these can nest
    - [{“one”:1, “o1”:{“a1”: [1,2,3.0], “a2”:[]}}]

- **JSON** = roughly this subset of Javascript

- The internal representation **varies**
  - In JS, 1 represents a 64 bit, IEEE floating point number
  - In Python’s json module, 1 represents a 32 bit integer in two’s complement
### External & Internal Representation

<table>
<thead>
<tr>
<th>Level</th>
<th>Data Unit Example</th>
<th>Information Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree/object/…</td>
<td><img src="image" alt="Tree Diagram" /></td>
<td>nesting, matching parentheses</td>
</tr>
<tr>
<td>Token</td>
<td>{&quot;name&quot;: {&quot;first&quot;: &quot;Bob&quot;,...}}</td>
<td>separator, types,…</td>
</tr>
<tr>
<td>Character</td>
<td>{&quot;name&quot;: {&quot;first&quot;: &quot;Bob&quot;,...}}</td>
<td>character encoding</td>
</tr>
<tr>
<td>Bit</td>
<td>10011010</td>
<td></td>
</tr>
</tbody>
</table>
JSON - XML example

```json
{"menu": {
    "id": "file",
    "value": "File",
    "popup": {
        "items": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
        ]
    }
}}
```

```xml
<menu id="file" value="File">
    <popup>
        <menuitem value="New" onclick="CreateNewDoc()" />
        <menuitem value="Open" onclick="OpenDoc()" />
        <menuitem value="Close" onclick="CloseDoc()" />
    </popup>
</menu>
```
JSON - XML example

```json
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {
          "value": "New",
          "onclick": "CreateNewDoc()"
        },
        {
          "value": "Open",
          "onclick": "OpenDoc()"
        },
        {
          "value": "Close",
          "onclick": "CloseDoc()"
        }
      ]
    }
  }
}
```

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

(order matters!

less different!)
Applications using XML

JSON!

Try it: http://jsonplaceholder.typicode.com
Twitter Demo

JSON and trees?

- JSON’s internal representation is
  - objects {} and arrays [] with nesting,
  - atoms
  - or similar structures in other programming languages
- Can we use it to represent tree-shaped data?
  - what are trees?
  - what kind of trees do we mean?
Interlude: Trees!
play a central role for SSD, JSON, XML,…. everything!
Trees come in different shapes!

Phylogenetic Tree of Life
Interlude: Abstract trees - nodes as strings!

A tree

A tree with strings as node names

A labelled tree over \{A,B,C\} (as node labels)

- so we can refer to nodes by names
- order matters!
  - the node 0,0 is different from 0,1

- so we can distinguish
  - a node from
  - a node’s label
The tree $T$ as a function:

$T(\varepsilon) = B$
$T(0) = A$
$T(1) = A$
$T(0,0) = B$
$T(0,1) = A$

The tree $T$ as a function:

A labelled $T$ tree over \{A,B,C\} (as node labels)

- so we can distinguish
  - a node from
  - a node’s label
Interlude: Abstract trees - nodes as strings!

- We use $\mathbb{N}$ for the non-negative integers (including 0)
- We use $\mathbb{N}^*$ for the set of all (finite) strings over $\mathbb{N}$
  - $\varepsilon$ is used for the empty string
  - 0,1,0 is a string of length 3
  - Each string stands for a node

- An alphabet is a finite set of symbols

- A tree $T$ over an alphabet $\Sigma$ is a mapping $T: \mathbb{N}^* \rightarrow \Sigma$ whose domain is
  - Finite, i.e., $T(n)$ is defined for only finitely many strings over $\mathbb{N}$
  - Each tree has only finitely many nodes
  - Contains $\varepsilon$, i.e., $T(\varepsilon)$ is defined
    - Each tree has a root $\varepsilon$
  - Prefix-closed, i.e., if $T(w,n)$ is defined, then $T(w)$ is as well
    - The predecessor $w$ of a node $(w,n)$ is in $T$
Interlude: Abstract trees - nodes as strings!

- Explanation:
  - the **strings** in the domain of T represent T’s nodes
  - \((w,n)\) is the successor of w,
  - \(T(w)\) is the label of w (as shown in picture)
  - we use nodes(T) for the (finite) domain of nodes in T

- **Is the following mapping T a tree? If yes, draw the tree T!**

\[
\Sigma = \{W, X, Y, Z\}\\
T(\epsilon) = X\\
T(0) = X\\
T(1) = X\\
T(2) = X\\
T(3) = Z\\
T(0,0) = Y\\
T(0,0,0) = Y\\
T(3,1) = Z
\]
Back to: JSON and trees?

- JSON’s internal representation is
  - objects {} and arrays [] with nesting,
  - atoms
  - or similar structures in other programming languages
- Can we use it to represent tree-shaped data?
  - what are trees?
  - what kind of trees do we mean?

```
{"root":{"label": "A", "children": 
  ["label": "B", "children":[]]},
  ["label": "C", "children": 
    ["label": "D", "children":[]],
    ["label": "E", "children":[]],
    ["label": "D", "children":[]]
  ]}
```

Although JSON IRs can be of other shapes & it’s not made for trees!
Self-Describing
Self-describing?!

- Some Data Models are said to be **self-describing**...what does this mean?
- Let’s compare to **CSV** (comma separated values):
  - each line is a **record**
  - commas separate **fields** (and no commas in fields!)
  - each record has the same number of fields
  - ...can you understand what this is about?

```
Tim, Morris, 2.32
Uli,     Sattler, 2.24
```
Self-describing?!

• One way of translating our example into JSON
  – ...can you understand what this is about?

```
Tim, Morris, 2.32
Uli, Sattler, 2.24
```

```json
[
  [
    "Tim",
    "Morris",
    "2.32"
  ],
  [
    "Uli",
    "Sattler",
    "2.21"
  ]
]
```
Self-describing?!

- One way of translating our example into JSON
  - ...can you understand what this is about?
  - ...or perhaps like this?

```
{
    "row1": {
      "att1": "Tim",
      "att2": "Morris",
      "att3": "2.32"
    },
    "row2": {
      "att1": "Uli",
      "att2": "Sattler",
      "att3": "2.24"
    }
}
```
Self-describing?!

- One way of translating our example into JSON
  - ...can you understand what this is about?
  - ...or perhaps like that???

```json
[{
  "row": {
    "att1": "Tim",
    "att2": "Morris",
    "att3": "2.32"
  }
}, {
  "row": {
    "att1": "Uli",
    "att2": "Sattler",
    "att3": "2.24"
  }
}]
```

Tim, Morris, 2.32
Uli, Sattler, 2.24
Self-describing?!

- Let’s consider a **self-describing CSV** (ExCSV)  
  - first line is **header** with **field names**  
  - ...can you understand what this is about?

- We could even **generically** translate such CSVs in JSON:

```
[  
  {   
    "name": "Tim",  
    "surname": "Morris",  
    "room": "2.32"  
  },  
  {   
    "name": "Uli",  
    "surname": "Sattler",  
    "room": "2.21"  
  }  
]
```

or, manually, even better:

```
{"addresses":  
  [{   
    "name": "Tim",  
    "surname": "Morris",  
    "room": "2.32"  
  },{   
    "name": "Uli",  
    "surname": "Sattler",  
    "room": "2.21"  
  }]}
```
Self-describing versus Guessability

- We can go a long way by **guessing**
  - CSV is *not easily* guessable
    - requires background knowledge
  - ExCSV is *more* guessable
    - still some guessing
    - could read the field tags and guess intent
    - had to guess the record type address
  - Guessability is tricky

- Is self-describing just being more or less guessable?

<table>
<thead>
<tr>
<th>Name</th>
<th>Surname</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bijan</td>
<td>Parsia</td>
<td>2.32</td>
</tr>
<tr>
<td>Uli</td>
<td>Sattler</td>
<td>2.24</td>
</tr>
</tbody>
</table>

{ "name": "Uli", "surname": "Sattler", "room": "2.21" }
Self-describing

*The Essence of XML* (Siméon and Walder 2003):
“From the **external representation** one should be able to
derive the corresponding **internal representation**.”

- **External**: the (Ex)CSV, JSON file/snippet, i.e., text!
- **Internal**:
  - e.g., the JS object, our application’s interpretation of the content
  - seems easy, but: in
    - `<room>2.32</room>` is “2.32” a string or a number?
    - `<height>2.32</height>` is “2.32” a string or a number?
    - …what should a your parser do?
    - …is 2.32 = 02.32 = 2.320? …is 2.32 > 2.32?
    - …is `{"first": "Tim","last": "Morris"} = {"last": "Morris","first": "Tim"} ?

- Are CSV, ExCSV, JSON self-describing?
Round-tripping

The Essence of XML (Siméon and Walder 2003):
“If one converts from an internal representation to the external representation and back again, the new internal representation should equal the old.”

- **External**: the (Ex)CSV, JSON file/snippet, i.e., text!
- **Internal**:  
  - e.g., the JS object, our application’s interpretation of the content  
  - seems easy, but: in  
    - `<room>2.32</room>` is “2.32” a string or a number?  
    - `<height>2.32</height>` is “2.32” a string or a number?  
    - …what should a your parser do?  
    - …is 2.32 = 02.32 = 2.320? …is 2.32 > 2.32?  
    - …is `{"first": "Tim","last": "Morris"}` = `{"last": "Morris","first": "Tim"}` ?
- Are CSV, ExCSV, JSON self-describing?
Self-describing

- Given
  1. a base format, e.g., ExCSV
  2. a/some specific document(s), e.g.,

- what suitable data structure can we extract?

  - CSV, ExCSV: tables, flat records, arrays, lists, etc.
  - JSON: labelled, ordered trees of (unbounded) depth!

- Clearly, you could parse *specific* CSV files into trees, but you’d need to use *extra*-CSV rules/information for that

- ...in this sense, XML can be said to be *more* self-describing than ExCSV

  **But** we still need to know whether “2.32” is a string or a number?

- **Schemas**!
Schemas: what are they?

A **schema** is a description

- of **DBs**: describes
  - tables,
  - their names and their attributes
  - keys, keyrefs
  - integrity constraints
Schemas: what are they?

A schema is a description

- of CSVs: describes
  - columns
  - their value range, i.e., which data goes where
  - ...

Diagram:
- A schema S
- Another schema S2
- All CSV files
- CSVs conforming to S
- CSVs conforming to S2
Schemas: what are they?

A **schema** is a description

- of **JSON documents**: describes
  - **structure**:
    - how objects/vectors are nested
    - which keys are required/possible
  - **data**: what values go where
    - some basic datatypes
    - with some restrictions
    - ...

A schema $S$ and another schema $S_2$:

- All JSON docs JSON valid wrt $S$
- JSON valid wrt $S_2$
Schemas: why?

• **RDBMS**
  – No database without schema
  – DB schema determines tables, attributes, names, etc.
  – Query optimization, integrity, etc.

• **CSVs, JSON (and XML)**
  – No schema *needed* at all!
  – “Legal” text snippets can be
    • parsed to yield data that can be
    • manipulated, queried, etc.
    • …but how, e.g., 2.32 = 2.320?
  – “Illegal”/broken snippets?!
What schemas can do

- A schema describes aspects of documents:
  - what’s **legal**: what a document can/may contain
  - what’s **expected**: what a document must contain
  - what’s **assumed**: default values

- Two **modes** for using a schema
  - **descriptive**:
    - describing documents
    - for other people
    - so that they know how to serialize their data
  - **prescriptive**:
    - prevent your application from using wrong documents
Benefits of an schema

- **Specification**
  - you document/describe/publish your format
  - so that it can be used across multiple implementations
- As **input** for applications
  - applications can do **error-checking** in a **format independent** way
    - checking whether an XML document conforms to a schema can be done by a **generic** tool (see CW2),
    - no need to be changed when schema changes
    - automatically!
Why schemas?

csv-parser for csvs, JSON-parser for JSON, …

needs to test for/handle errors due to text having been in wrong format

text in some format

parser for this format

Serializer

Standard API

your application
Because schemas are great!

- For communication between developers:
  - my schema shows you what I expect/accept
  - your schema shows me what I should produce
- in a commonly understood format
- supported by tools: **validators** or **schema-aware parsers**
  - applications can do **error-checking** in a **format independent** way

```
Validates text against (any) schema
```

```
No need to test for errors - just handle validator’s error message!
```

```
Schema
  └── text in some format

  └── schema-aware parser

      Standard API

      Serializer

      your application
```
CSVW
Your 1st Schema Language
CSV on the Web

• The W3C Working Group CSV on the Web has developed a way to
  – describe CSV (what goes where?)
  – on the web
  – in a JSON object
• for developers
  – so that they know what to export/build/share
• for tools
  – so that we can validate CSVs
• now a recommendation
  – https://www.w3.org/TR/tabular-metadata/
• see
  – its primer https://www.w3.org/TR/2016/NOTE-tabular-data-primer-20160225/
  – python parser/validators
    • https://github.com/sebneu/csvw-parser
    • https://pypi.org/project/csvw/
  – its community group https://www.w3.org/community/csvw/
CSVW - an example

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "url": "countries.csv",
  "tableSchema": {
    "columns": [{
      "titles": "country",
      "datatype": "string" },{
      "titles": "country group",
      "datatype": "string" },{
      "titles": "name (en)",
      "datatype": "string" },{
      "titles": "name (fr)",
      "datatype": "string" },{
      "titles": "name (de)",
      "datatype": "string" },{
      "titles": "latitude",
      "datatype": "number" },{
      "titles": "longitude",
      "datatype": "number"
    }]
  }
}
```

<table>
<thead>
<tr>
<th>country</th>
<th>country group</th>
<th>name (en)</th>
<th>name (fr)</th>
<th>name (de)</th>
<th>latitude</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>eu</td>
<td>Austria</td>
<td>Autriche</td>
<td>Österreich</td>
<td>47.6965545</td>
<td>13.34598005</td>
</tr>
<tr>
<td>be</td>
<td>eu</td>
<td>Belgium</td>
<td>Belgique</td>
<td>Belgien</td>
<td>50.501045</td>
<td>4.47667405</td>
</tr>
<tr>
<td>bg</td>
<td>eu</td>
<td>Bulgaria</td>
<td>Bulgarie</td>
<td>Bulgen</td>
<td>42.72567375</td>
<td>25.48232186</td>
</tr>
</tbody>
</table>
CSVW - Datatypes

CSVW supports *built-in datatypes* plus XML Schema (XSD) Data Types

- plus XSD’s mechanisms to define new datatypes by restricting base types
- plus naming & annotating

```
"datatype": {
  "dc:title": "Star Rating",
  "dc:description": "A star rating between 1 and 5."
  "base": "integer",
  "minimum": "1",
  "maximum": "5"
}
```

```
"datatype": {
  "dc:title": "Country Code",
  "dc:description": "Country codes as specified in ISO 3166."
  "base": "string",
  "format": "[a-z]{2}"}
```
CSVW supports *built-in datatypes* plus XML Schema (XSD) Data Types

- plus XSD’s mechanisms to define new datatypes by restricting base types
- plus naming & annotating
- plus some more

```json
"columns": [
  {
    "titles": "country",
    "datatype": {
      "base": "string",
      "minLength": "3",
      "maxLength": "128"},
    "required": true},
  {
    "titles": "country group",
    "datatype": {
      "base": "string",
      "minLength": "2",
      "maxLength": "2"},
    "required": true},
  {
    "titles": "latitude",
    "datatype": {
      "base": "number",
      "minimum": "-90",
      "maximum": "90"},
    "required": true},
  {
    "titles": "longitude",
    "datatype": {
      "base": "number",
      "minimum": "-180",
      "maximum": "180"},
    "required": true},
  ...
]
```
Why is this cool?

- CSVW Schema
- a CSV file
- schema-aware parser
- Serializer
- Standard API
- your application

Validates CSV against this schema

No need to test for errors - just handle validator’s error message!
Validation and Being Valid

- A CSV $D$ is valid wrt/against a schema $S$ if $D$ satisfies all constraints in $S$.
- Checking whether $D$ is valid wrt $S$ is called validation: you could
  - do this by hand or
  - use a schema-aware/validating parser for it or
  - use a validator
- **Note**: $D$ can be valid wrt $S$ without any validation having taken place!
Just the tip of an iceberg!

You can use a CSVW to describe

- a group of tables
- uniqueness and (foreign) keys constraints
- defaults values for cells
- column titles, possibly with language code
- use regular expressions to describe column titles or values
- use various vocabularies, e.g., Dublin Core
- units of measures that go with column
- how to link CSVs to their schema:
  - use a CSV with a Link header or
  - put in same directory, following naming conventions: MyTable.csv and MyTable-metadata.json
  - describe it in / .well-known/csvm file on your server
  - ...

Interlude: Regular Expressions
Regular Expressions

- a standard concept to describe sets of strings
- used in (almost all) programming or schema languages
- sometimes in different syntaxes
- but the principles are the same
Regular expressions: definition

- Given a set of **symbols** \( N \), the set of **regular expressions** \( \text{regexp}(N) \) over \( N \) is the smallest set containing
  - the empty string \( \varepsilon \) and all symbols in \( N \) and
  - if \( e_1 \) and \( e_2 \in \text{regexp}(N) \), then so are
    - \( e_1, e_2 \) (concatenation)
    - \( e_1 | e_2 \) (choice)
    - \( e_1^* \) (repetition)

- Given a regular expression \( e \), a string \( w \) **matches** \( e \),
  - if \( w = \varepsilon = e \) or \( w = n = e \) for some \( n \) in \( N \), or
  - if \( w = w_1w_2 \) and \( e = (e_1, e_2) \) and
    \( w_1 \) matches \( e_1 \) and \( w_2 \) matches \( e_2 \), or
  - if \( e = (e_1 | e_2) \) and \( w \) matches \( e_1 \) or \( w \) matches \( e_2 \)
  - if \( w = \varepsilon \) and \( e = e_1^* \)
  - if \( w = w_1w_2\ldots w_n \) and \( e = e_1^* \) and each \( w_i \) matches \( e_1 \)
Regular expressions

- Hence we can use
  - $e^+$ as abbreviation for $(e,e^*)$
  - $e?$ as abbreviation for $(e|\varepsilon)$

Let’s test our understanding via some Kahoot quiz: go to kahoot.it
JSON Schema
Your 2nd Schema Language
Is JSON edging towards SQL completeness?

• Do we have (even post-facto) schemas?
  – Historically, mostly code
  – But there have been schema proposals:
    • JSON-Schema
      – try it out: http://jsonschema.net/#/
    • JSON-Schema Draft 7
      – Rather simple!
      – Simple patterns
        • Types on values (but few types!)
        • Some participation/cardinality constraints
          • allOf, oneOf,..
        • Lexical patterns
          – Email addresses!
Example

- [http://json-schema.org/example1.html](http://json-schema.org/example1.html)

```json
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "name": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    }
  },
  "required": ["id", "name", "price"]
}
```

I am a JSON Schema!
Docs start with an object which can have these props
Types, possibly with further restrictions
Must have all of these!
Types supported by JSON Schema

• Strings, can be restricted via
  – max/min length or
  – regular expressions
  – plus built-in formats (date-time, time, date,…)

• Numbers
  – integer
  – number for integer or floats
  – can be restricted via minimum, exclusiveMinimum,…multipleOf,…

• Objects {}, can be restricted via
  – their size using minProperties, …
  – dependencies
  – patterns,…

• Arrays [] can be restricted via size, type of (all, some, tuple) content
• Boolean
• Null

Recursively: we can name & re-use sub-schemas

Via enumerations

With comments, descriptions…
Why is that useful?

Validating "{"id": 10, "name": "AAAA"}" with our schema results in
- an error
- with helpful error message "object has missing required properties (["price"])"

Validating "{"id": 10, "name": "AAAA", "price": -2}" with our schema results in
- an error
- with helpful error message "numeric instance is lower than the required minimum (minimum: 0, found: -2)"

E.g., https://github.com/java-json-tools/json-schema-validator

Validates text against (any) schema
No need to test for errors - just handle
More Examples: further restrictions of atomic types

{
  ..
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer",
      "minimum": 10
    },
    "name": {
      "description": "Name of the product",
      "type": "string",
      "maxLength": 28,
      "minLength": 4
    },
    "price": {
      "type": "number",
      "minimum": 0,
      "maximum": 1000
    }
  },
  "required": ["id", "name", "price"]
}
More Examples: nesting

```json
{
  ...
  "properties": {
    "id": {
      "description": "The unique identifier for a person",
      "type": "integer",
      "minimum": 10,
    },
    "name": {
      "description": "Name of the person",
      "type": "object",
      "properties": {
        "first": {"type": "string"},
        "last": {"type": "string"},
        "others": {"type": "string"}
      },
      "required": ["first", "last"],
      "additionalProperties": false
    },
  },
  "required": ["id", "name"]
}
```
More Examples: named sub-schemas, structuring

{"$schema": "http://json-schema.org/draft-04/schema#",
 "definitions": {
   "GoodString": {
     "description": "Strings for Names",
     "type": "string",
     "pattern": "[A-Z][a-z]+" },
   "title": "Product",
   "description": "Uli's example",
   "type": "object",
   "properties": {
     "id": {"description": "The unique identifier for a person",
            "type": "integer" },
     "name": {"type": "object",
               "properties": {
                 "first": {"$ref": "#/definitions/GoodString"},
                 "last": {"$ref": "#/definitions/GoodString"},
                 "others":{"$ref": "#/definitions/GoodString"} },
                 "required": ["first", "last"],
                 "additionalProperties": false },
     "age": {"type": "number",
              "minimum": 0,
              "maximum": 100 } },
     "required": ["id", "name","age"]}
Why is that useful?

Validating \{"id": 10, "name": \{"first": "Bob", "last": "Smith", "others": "Fee"\}, "age": 2\}" with our schema results succeeds.

Validating \{"id": 10, "name": \{"first": "bob", "other": "Fee"\}, "age": 2\}" with our schema results in:
- an error
- with (long) helpful error message

E.g., [https://github.com/java-json-tools/json-schema-validator](https://github.com/java-json-tools/json-schema-validator) or [https://pypi.org/project/jsonschema/](https://pypi.org/project/jsonschema/)

- Validates text against (any) schema
- No need to test for errors - just handle
My favourite example: trees!

Describing trees requires recursion!

```json
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "description": "A schema for trees",
    "definitions": {
        "node": {
            "type": "object",
            "properties": {
                "label": {
                    "type": "string",
                    "minLength": 1,
                    "maxLength": 20
                },
                "childnodes": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/node"
                    }
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "root": {
            "$ref": "#/definitions/node"
        }
    },
    "required": ["root"]
}
```
Validation and Being Valid

- A JSON text $D$ is valid wrt/against a schema $S$ if $D$ satisfies all constraints in $S$.
- Checking whether $D$ is valid wrt $S$ is called **validation**: you could
  - do this by hand or
  - use a schema-aware/validating parser for it or
  - use a validator
- **Note**: $D$ can be valid wrt $S$ without any validation having taken place!

![Diagram showing relationships between JSON, schemas, and validation](image-url)
Just the tip of an iceberg!

- You can use **regular expression** for restrictions of strings
  - for general strings
  - or describing (strings for) keys in arrays
- What can/can’t you say in JSON Schema?
  - can you have *either* first or last name?
- How to structure?
- How to link instance documents to JSON schema (via “$ref”)?
- How strict do you want to be
  - for your application
- Can you catch **all errors**?
  - wrong Postcode/streetname combinations?
  - wrong age/DoB combinations?
- Try it out at https://www.jsonschemavalidator.net/
- …and explore it in your coursework!
JSON *Databases*?

- NoSQL “movement”
  - Originally “throw out features”
    - Still quite a bit
  - Now, a bit of umbrella term for semi-structured databases
    - So XML counts!
  - Some subtypes:
    - Key-Value stores
    - Document-oriented databases
    - Graph databases
    - Column databases
- Some support JSON as a layer
  - E.g., BaseX
- Some are “JSON native”
  - MongoDB
  - CouchDB
Phew - Summary of today

We have seen many things - you’ll deepen your understanding in coursework:

**Tree data models:**

1. Data Structure formalisms: JSON
2. Schema Languages: JSON Schema, CSVW
3. Data Manipulation: Python

**General concepts:**

- Semi-structured data
- Self-Describing
- Trees
- Regular Expressions
- Internal & External Representation, Parsing & Serialising
- Validation, valid, …
- Format
Next: Coursework Old & New

- Quiz
- Short essay: think more about Self-Describing
- M2: design your own `format` for addresses, JSON-based, and create a JSON Schema for it
- CW2: interact with (nested) arithmetic expressions in JSON