The Forms of Descriptions

Ryan Shaw, University of North Carolina at Chapel Hill

This is a revised version of "The Forms of Resource Descriptions," in Robert J. Glushko, ed., The Discipline of Organizing*, chapter 8, page 283–324. MIT Press, Cambridge, Massachusetts, 2013.*

The problem of how to form descriptions can be viewed from two perspectives. From one perspective, descriptions are *things that are used* by both people and computational agents. From this perspective, choosing the form of descriptions is a kind of design. This is easy to see for certain kinds of descriptions, like the signs and maps found in physical environments such as airport terminals, public libraries, and malls. In these spaces, descriptions are quite literally designed to help people orient themselves and find their way. But any kind of description, not just ones embedded in the built environment, can be viewed as a designed object. Designing an object involves making decisions about how it should be structured so that it can best be used for its intended purpose. From a design perspective, choosing the form of a description means making decisions about its *structure*.

From another perspective, however, creating descriptions is a kind of *writing*. I may describe something to you verbally, but that description would not be very useful for an organizing system. Organizing systems need persistent descriptions, and that means they need to be written down or recorded in some way. In that sense, choosing the form of a description means making decisions about *notation* and *syntax*.

Modern Western culture tends to make a sharp distinction between designing and writing, but there are areas where this distinction breaks down, and the creation of descriptions in organizing systems is one of them. Here I use designing and writing as two lenses for looking at the spectrum of options for structuring descriptions, and the kinds of syntaxes available for writing those descriptions down.

1 Designing descriptions

Choosing how to structure descriptions is a matter of making design decisions. Hopefully, these decisions are made not randomly, but in order to solve specific problems, serve specific purposes, or bring about some desirable property in the descriptions. Most of these decisions are specific to a *domain*: the particular context of application for the organizing system being designed and the kinds of interactions it will enable. Making these kinds of context-specific decisions results in a model of that domain.

But as many people have built similar kinds of descriptions over time, they've faced similar problems, had similar purposes, and desired similar properties. Unsurprisingly, they have

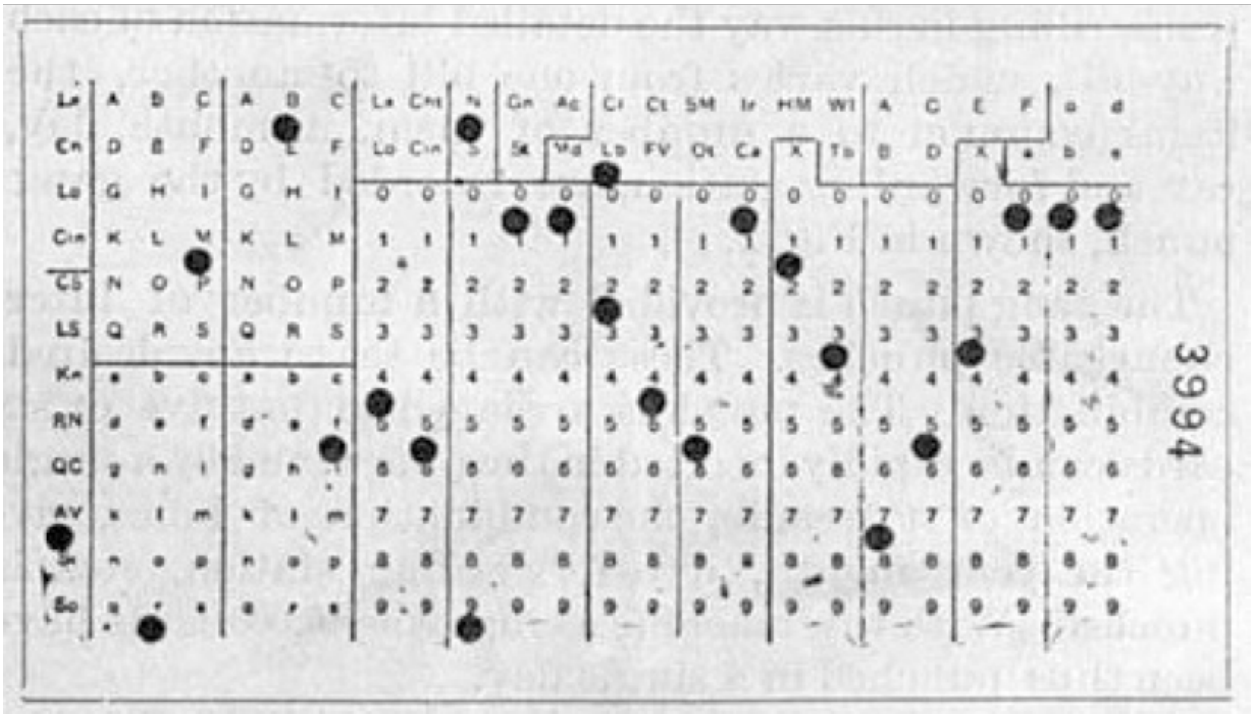

Image 1. A Hollerith punched card

converged on some of the same decisions. When common sets of design decisions can be identified that are not specific to any one domain, they often become formally recognized and designed into standard formats and architectures for creating organizing systems. These formally recognized sets of design decisions are known as **abstract models** or **metamodels**. Metamodels describe structures commonly found in descriptions and other information objects, regardless of the specific domain. While any designer of an organizing system will usually create a model of her specific domain, she usually won't create a new metamodel but will instead make choices from among the metamodels that have been formally recognized and incorporated into existing standards. Reusing standard metamodels can bring great economical advantages, as developers can reuse tools designed for and knowledge about these metamodels, rather than having to start from scratch. The following sections present some common kinds of structures used as the basis for metamodels. But first, consider a concrete example of how the structure of descriptions supports or inhibits particular uses.

During World War II, a British chemist named W. E. Batten developed a system for organizing patents.[1] The system consisted of a language for describing the product, process, use, and apparatus of a patent, and a way of using punched cards to record these descriptions. Batten used cards printed with matrices of 800 positions (see Image 1). Each

[1] This discussion of Batten's cards is based on Lancaster, *Information Retrieval Systems*, 1968, 28–32.

card represented a specific value from the vocabulary of the description language, and each position corresponded to a particular patent. To describe patent #256 as covering *extrusion of polythene to produce cable coverings*, one would first select the cards for the values *polythene*, *extrusion*, and *cable coverings*, and then punch each card at the 256th position. The description of patent #256 would thus extend over these three cards.

The advantage of this structure is that to find patents covering *extrusion of polythene* (for any purpose), one needs only to select the two cards corresponding to those values, lay one on top of the other, and hold them up to a light. Light will shine through wherever there is a position corresponding to a patent described using those values. Patents meeting a certain description are easily found due to the structure of the cards designed to describe the patents.

Of course, this system has clear disadvantages as well. Finding the concepts associated with a particular patent is tedious, because every card must be inspected. Adding a new patent is relatively easy as long as there is an index that allows the cards for specific concepts to be located quickly. However, once the cards run out of space for punching holes, the whole set of cards must be duplicated to accommodate more patents: a very expensive operation. Adding new concepts is potentially easy: simply add a new card. But to find existing patents using the new concept, all the existing patents would have to be re-examined to determine whether their positions on the new card should be punched: also an expensive operation.

The structure of Batten's cards supported rapid selection of patents given a partial description. The kinds of structures surveyed in the following sections are not quite so elaborate as Batten's cards. But like the cards, each kind of structure supports more efficient mechanical execution of certain operations, at the cost of less efficient execution of others. The structures of descriptions enable or inhibit particular ways of interacting with those descriptions, just as the descriptions themselves enable or inhibit particular ways of interacting with the described things.

1.1 Kinds of structures

Sets, lists, dictionaries, trees, and graphs are kinds of structures that can be used to form descriptions. Each of these kinds is actually a family of related structures. These structures are *abstractions*: they describe formal structural properties in a general way, rather than specifying an exact physical or textual form. Abstractions are useful because they help us to see common properties shared by different specific ways of formatting information. By focusing on these common properties, one can more easily reason about the operations that different forms support and the affordances that they provide, without being distracted by less relevant details.

1.1.1 Blobs

The simplest kind of structure is no structure at all. A description with no structure is sometimes referred to as a blob. Consider the following description of a book: *Sebald's novel uses a walking tour in East Anglia to meditate on links between past and present, East and West*.[2] This description is a blob of text, without structure. Or, more precisely, it has structure, but that structure is the structure of the English language. Readers of English can interpret this as a description of the subject of the book, but to do so mechanically (i.e. using a machine such as a computer) is more difficult. On the other hand, such a description is relatively easy to create, as the describer can simply use natural language.

A blob needn't be a blob of text. It could be a photograph of something, or a recording of a spoken description of that thing. Like blobs of text, blobs of pixels or sound have structure that is easy for humans to comprehend. But considered from the perspective of how they support or inhibit mechanical or computational operations, these blobs are unstructured.

1.1.2 Sets

A blob lacks clearly defined parts. The simplest way to give a description parts is to make it a **set**. For example, the description above might be reformulated as a set of terms: *Sebald*, *novel, East Anglia, walking, history*. Much of the meaning has been lost by doing this, but something has been gained: one now can easily identify *Sebald* and *walking* as separate items in the description. This makes it easier to find, for example, all the descriptions which include the term *walking*. (Note that this is different from simply searching through blob-of-text descriptions for the word *walking*. When treated as a set, the description *Fiji, fire walking, memoir* does not include the term *walking*, though it does include the term *fire walking*.)

Sets make it easy to find intersections among descriptions. Sets are also easy to create. Consider "folksonomies," organizing systems in which non-professionals create descriptions. In these systems, descriptions are structured as sets of "tags." Once can specify a set of tags to find things having descriptions that intersect at those tags. This is more valuable if the tags come from a controlled vocabulary, making intersections more likely. But enforcing vocabulary control adds complexity to the description process, so a balance must be struck between maximizing potential intersections and making description as simple as possible.

A set is a type or class of structure. Different kinds of sets can be defined by introducing **constraints**. For example, one might introduce the constraint that a list has a maximum number of items. Or one might constrain a set to always have the same number of items,

---

[2] Roberta Silman, "In the Company of Ghosts," *New York Times*, July 26, 1998. http://www.nytimes.com/books/98/07/26/reviews/980726.26silmant.html

giving us a fixed-size list. Constraints can also be removed. Sets don't contain duplicate items (think of a tagging system in which it doesn't make sense to assign the same tag more than once to the same thing). If this constraint is removed, the result is a different structure known as a *bag* or *multiset*.

1.1.3 Lists

Different constraints are also what distinguish sets from lists. Like a set, a **list** is a collection of items. But lists add an additional constraint: their items are *ordered*. If you were designing a tagging system in which it was important that the order of the tags be maintained, you would want to use lists, not sets. Unlike sets, lists may contain duplicate items. In a list, two items that are otherwise the same can be distinguished by their position in the ordering, but in a set this is not possible.

Constraints can be introduced to define different kinds of lists, such as fixed-length lists. If one constrains list to contain only items which are themselves lists, and further specifies that these contained lists do not themselves contain lists, then the result is a table (a list of lists of items). A spreadsheet is basically a list of lists.

1.1.4 Dictionaries

One major limitation of lists and sets is that, although items can be individually addressed, there is no way to distinguish among those items except by comparing their values (or, in a list, their positions in the ordering). In a set of tags like *Sebald*, *novel, East Anglia, walking, history*, for example, one cannot easily tell that *Sebald* refers to the author of the book while *East Anglia* and *walking* refer to what it is about. One way of addressing this problem is to break each item in a set into two parts: a **property** and a **value**. So, for example, the simple set of tags might become *author: Sebald, type: novel, subject: East Anglia, subject: walking, subject: history*. Now *author*, *type*, and *subject* are the properties, and the original items in the set are the values.

This kind of structure is called a **dictionary**, also known as a *map* or an *associative array.* A dictionary is a set of property-value pairs or *entries*. It is a set of entries, not a list of entries, because the pairs are not ordered and because each entry must have a unique key. Note that this specialized meaning of "dictionary" is different from the more common meaning of "dictionary" as a list of terms accompanied by sentences that define them. The two meanings are related, however. Like a "real" dictionary, a dictionary structure allows us to easily find the value (such as a definition) associated with a particular property or *key* (such

as a word).[3] But unlike a real dictionary, which orders its keys alphabetically, a dictionary structure doesn't specify an order for its keys.

Dictionaries are found everywhere in descriptions. Structured descriptions entered using a form are easily represented as dictionaries, where the labels of the form items are the properties and the data entered are the values. Tabular data with a "header row" can be thought of as a set of dictionaries, where the column headers are the properties for each dictionary, and each row is a set of corresponding values. Dictionaries are also a basic type of data structure found in nearly all programming languages (where they usually are referred to as associative arrays).

One can introduce or remove constraints to define specialized types of dictionaries. A sorted dictionary adds an ordering over entries; in other words, it is a list of entries rather than a set. A *multimap* is a dictionary in which multiple entries may have the same key.

1.1.5 Trees

In dictionaries as they are commonly understood, the values are term definitions. But in dictionaries understood as abstract sets of property-value pairs, the values can be anything at all. In particular, the values can themselves be dictionaries. When a dictionary structure has values that are themselves dictionaries, the dictionaries are *nested*. Nesting is very useful for descriptions that need more structure than what a (non-nested) dictionary can provide.

Figure 1 shows an example. At the top level there is one dictionary with a single entry having the property *a*. The value associated with *a* is a dictionary consisting of two entries, the first having property *b* and the second having property *c*. The values associated with *b* and with *c* are also dictionaries.

If dictionaries are nested, and the "top" dictionary (the one that contains all the others) has only one entry, then it forms a **tree** structure. Figure 2 shows the same properties and values as Figure 1, this time arranged to make the tree structure more visible. Trees consist of **nodes** (the letters and numbers in Figure 2) joined by **edges** (the arrows). Each node in the tree with a circle around it is a property, and the value of each property consists of the nodes below (to the right of) it in the tree. A node is referred to as the *parent* of the nodes below it, which in turn are referred to as the *children* of that node. The edges show these "parent of" relationships between the nodes. The node with no parent is called the *root* of the tree. Nodes with no children are called *leaf* nodes.

As with the other types of structures, different kinds of trees can be defined by introducing different types of constraints. Nested dictionaries are a kind of tree in which child nodes do

[3] Going the other direction is not so easy, however: just as real dictionaries don't support finding a word given a definition, neither do dictionary structures support finding a key given a value.

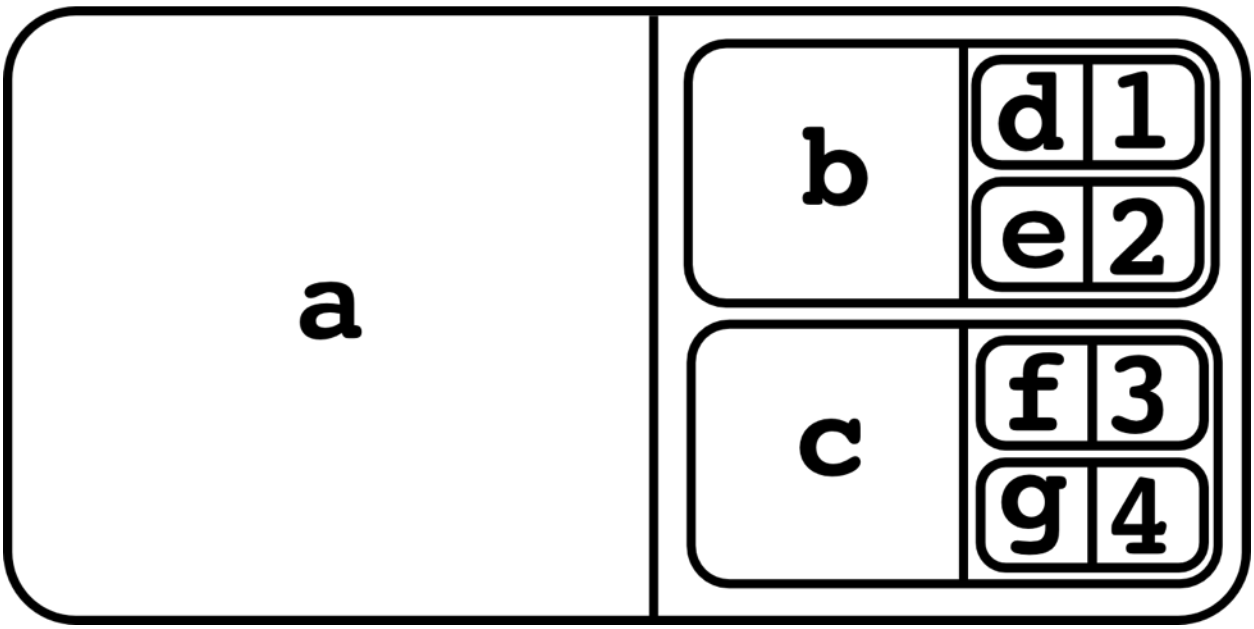

Figure 1. Four nested dictionaries

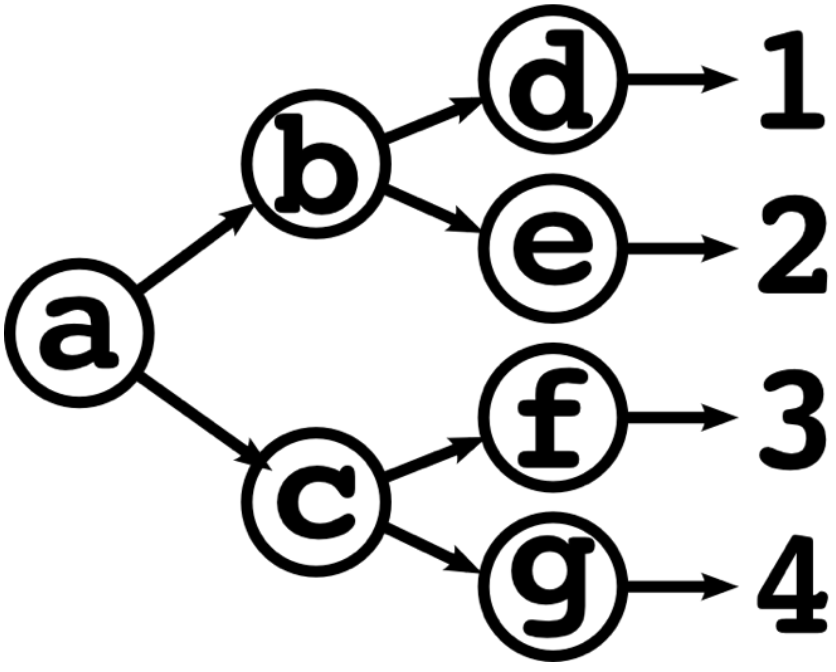

Figure 2. A tree of properties and values

not have any order (because dictionary entries do not have an ordering). But it is common to add an ordering constraint to tree structures. Ordering is important for descriptions, since without ordering it is impossible to (for example) list multiple authors while guaranteeing that the order of authors will be maintained. For example, the metamodel for XML documents is a kind of tree called the XML Information Set or Infoset.

The Infoset defines a specific kind of tree structure by adding very specific constraints, including ordering of child nodes, to the basic definition of a tree. Similarly, Figure 2 actually depicts a highly constrained kind of tree, in which all non-leaf nodes are properties, and all leafs are values. It's possible to select different constraints to define different kinds of trees. For example, one might define a tree in which every node has both a property and a value. Trees exist in a large variety of flavors, but they all share a common topology: the edges between nodes are directed (one node is the parent and the other is the child), and every node except the root has exactly one parent.

Trees provide a way to group statements describing different but related things. For example, consider the dictionary-structured description in Example 1. The dictionary groups together four property-value pairs describing a particular book. (The arrows are simply a schematic way to indicate property-value relations.) But really the first two entries are not describing the book; they are describing the book's author. So, it would be better to group those two statements by nesting the entries describing the author within the book description, creating a tree structure (Example 2).

Using a tree works well in this case because the book can be treated as the primary thing being described, making it the root of the tree, with the author description as a "branch." It's also possible to make the author the primary thing, resulting in a tree like the one in Example 3.

Note that in this dictionary, the value of the *books authored* property is a *list* of dictionaries. Making the author the primary or root thing allows us to include multiple book descriptions in the tree (but might make it more difficult to describe books having multiple authors). A tree is a good choice for structuring descriptions if a primary thing can be identified. In some cases, however, one wants to connect descriptions of related things without having to designate one as primary. In such cases, a more flexible data structure is needed.

```
author given names → Winfried Georg
author surname → Sebald
title → Die Ringe des Saturn
pages → 371
```

Example 1. Description structured as a dictionary

```
author →
       given names → Winfried Georg
       surname → Sebald
title → Die Ringe des Saturn
pages → 371
```

Example 2. Nesting an author description within a book description

```
given names → Winfried Georg
surname → Sebald
books authored →
    1. title → Die Ringe des Saturn
       pages → 371
    2. title → Austerlitz
       pages → 416
```

Example 3. Nesting book descriptions within an author description

1.1.6 Graphs

Suppose one were describing two books, where the author of one book is the subject of the other (Example 4).

```
1. author → Mark Richard McCulloh
   title → Understanding W. G. Sebald
   subject → Winfried Georg Sebald

2. author → Winfried Georg Sebald
   title → Die Ringe des Saturn
```

Example 4. Two related descriptions

By looking at these descriptions, one can guess the relationship between the two books, but that relationship isn't explicitly represented in the structure: there are two separate dictionaries, and one has to infer the relationship by matching property values. It's possible that this inference could be wrong: there might be two people named *Winfried Georg Sebald*. How can these descriptions be structured to explicitly represent the fact that the *Winfried Georg Sebald* that is the subject of the first book is the same *Winfried Georg Sebald* who authored the second?

One possibility would be to make *Winfried Georg Sebald* the root of a tree, similar to the approach taken in Example 3, adding a *books about* property alongside the *books authored* one. This solution would work fine if people were the primary things, and it thus made sense to structure descriptions around them. But suppose that you wanted to structure your descriptions around books, perhaps because you were using a vocabulary that took this perspective (with properties such as *author* and *subject* rather than *books authored* and *books about*). Being tied to particular structure should not dictate the organizational perspective one can take, as Batten's cards did. Instead, structures should be consciously chosen to suit one's organizational perspective. How can this be done?

If the two book descriptions are structured as trees, the two branches (`subject` and `author`) that share a value can be joined. The result is no longer a tree, because now there is a node with more than one parent (Figure 3). The structure in Figure 3 is a **graph**. Like a tree, a graph consists of a set of nodes connected by edges. These edges may or may not have a direction. If they do, the graph is referred to as a *directed* graph. If a graph is directed, it may be possible to start at a node and follow edges in a path that leads back to the

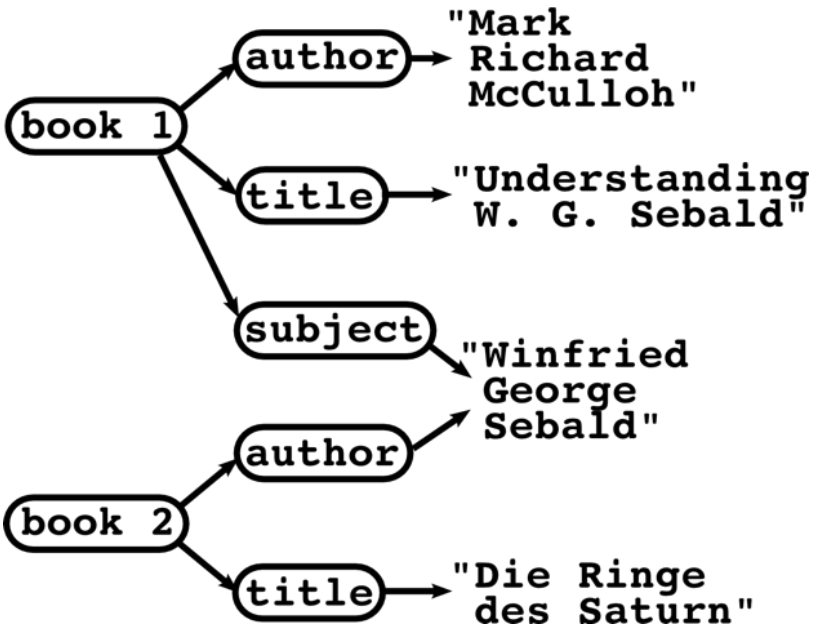

Figure 3. Descriptions linked into a graph

starting node. Such a path is called a *cycle*. If a directed graph has no cycles, it is referred to as an *acyclic* graph.

A tree is just a more constrained kind of graph. Trees are directed graphs because the "parent of" relationship between nodes is asymmetric: the edges are arrows that point in a certain direction. Furthermore, trees are acyclic graphs, because if you follow the directed edges from one node to another, you can never encounter the same node twice. Finally, trees have the constraint that every node (except the root) must have exactly one parent.[4] In Figure 3, this constraint has been violated by joining the two book trees. The graph that results is still directed and acyclic, but because the *Winfried George Sebald* node now has two parents, it is no longer a tree.

Graphs are very general and flexible structures. Many kinds of systems can be conceived of as nodes connected by edges: stations connected by subway lines, people connected by friendships, decisions connected by dependencies, and so on. Relationships can be modeled in different ways using different kinds of graphs. For example, if friendship is treated as symmetric, one might use an undirected graph, but if friendship is asymmetric (I might consider you a friend without you reciprocating), a directed graph is more appropriate.

[4] Technically, what is described here is referred to as "rooted tree" by mathematicians, who define trees more generally. Since trees used as data structures are always rooted trees, I do not make the distinction.

Often it is useful to treat a graph as a set of pairs of nodes, where each pair may or may not be directly connected by an edge. Many approaches to characterizing structural relationships among things are based on modeling the related things as a set of pairs of nodes, and then analyzing patterns of connectedness among them. Being able to break down a graph into pairs is also useful when structuring descriptions as graphs.

1.2 Comparing metamodels: JSON, XML and RDF

In the previous section I surveyed the various *kinds* of metamodels used to structure descriptions. In this section I look at some *specific* metamodels. A detailed comparison of the affordances of different metamodels is beyond the scope of this overview. Here I simply take a brief look at three popular metamodels—JSON, XML, and RDF—in order to show how they further specify and constrain the more general kinds of metamodels introduced above.

1.2.1 JSON

JavaScript Object Notation (JSON) is a textual format for exchanging data that borrows its metamodel from the JavaScript programming language. Specifically, the JSON metamodel consists of two kinds of structures found in JavaScript: lists (called *arrays* in JavaScript) and dictionaries (called *objects* in JavaScript). Lists and dictionaries contain values, which may be strings of text, numbers, Booleans (true or false), or the null (empty) value. Again, these types of values are taken directly from JavaScript. Lists and dictionaries can be values too, meaning lists and dictionaries can be nested within one another to produce more complex structures such as tables and trees.

Lists, dictionaries, and a basic set of value types constitute the JSON metamodel. This metamodel is just a subset of JavaScript, so the JSON metamodel is very easy to work with in JavaScript. Since JavaScript is the only programming language that is standardly available in all Web browsers, JSON has become a popular choice for developers who need to work with data and descriptions on the Web (see section 3.2). Furthermore, almost all modern programming languages provide data structures and value types equivalent to those provided by JavaScript. So, data represented as JSON is easy to work with in any programming language, not just JavaScript.

1.2.2 XML

The **eXtensible Markup Language (XML) Infoset** metamodel presents a strong contrast to JSON. Where the JSON metamodel is derived from data structures found in programming languages, the Infoset is derived from data structures used for document markup. These markup structures—**elements** and **attributes**—are optimized not for

programmatically manipulating data but for imposing structure on text. The Infoset is a tree structure, where each node of the tree is defined to be an *information item* of a particular type. Each information item has a set of type-specific properties associated with it. At the root of the tree is a document item, which has exactly one element item as its child. An element item has a set of attribute items, and a list of child nodes. These child nodes may include other element items, or they may be character items (see section 2.1 below for more on characters).

Figure 4 shows how the Infoset might be used to structure part of a description of an author and his works. This example demonstrates how element items might be used to model the domain of the description, by giving them names such as author and title. The character items that are the children of these elements hold the content of the description: author names, book titles, and so on. Attribute items are used to hold auxiliary information about this content, such as its language.

This example also demonstrates how the XML Infoset supports "mixed content" by allowing element items and character items to be "siblings" with the same parent element. It is this structural feature, combined with the fact that child nodes in the Infoset are ordered, that makes it possible for XML documents to function both as human reader-oriented, textual documents and as structured data formats. In this case, the Infoset structure allows us to specify that the book description can be displayed as a line of text consisting of the original title and the translated title in parentheses. The elements and attributes are used to

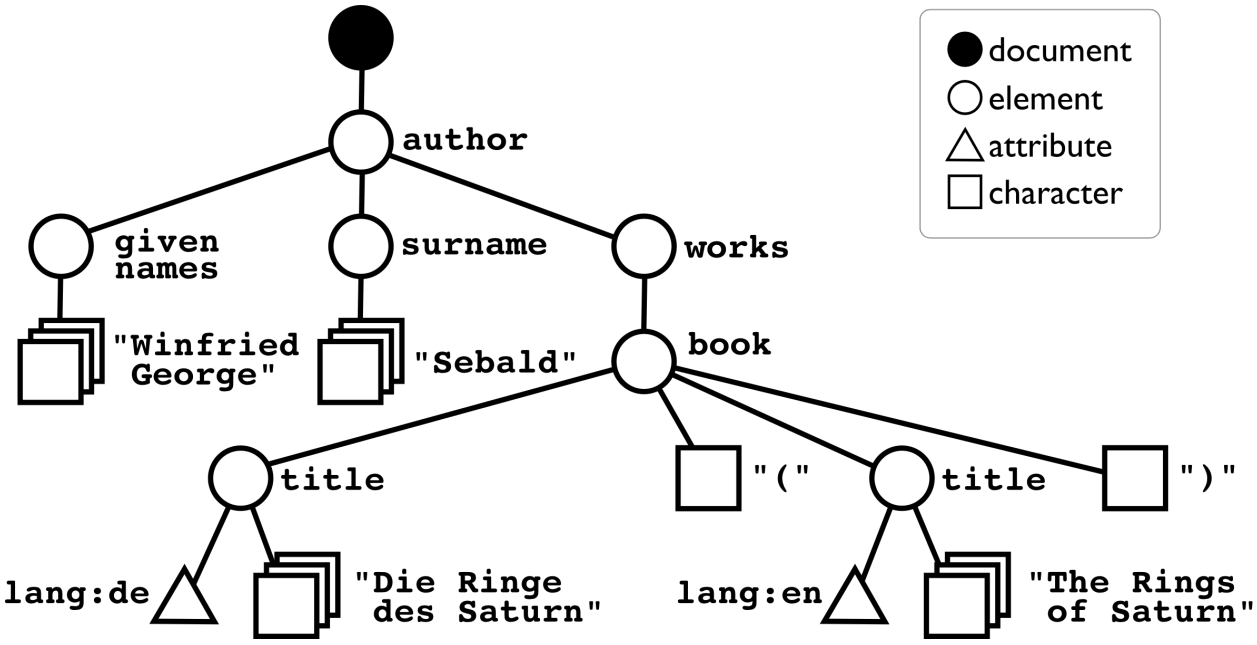

Figure 4. A description structure conforming to the XML Infoset metamodel

indicate that this line of text consists of two titles written in different languages, not a single title containing parentheses.

1.2.3 RDF

In Figure 3, the description was structured as a graph by treating things, properties and values as nodes, with edges reflecting their combination into descriptive statements. However, a more common approach is to treat things and values as nodes, and properties as the edges that connect them. Figure 5 shows the same description as Figure 3, this time with properties treated as edges. Figure 5 roughly corresponds with the particular kind of graph metamodel defined by the **Resource Description Framework (RDF)**.

A graph can be viewed as as a set of pairs of nodes, where each pair may be connected by an edge. Similarly, each component of the description in Figure 5 can be viewed as a pair of nodes (a thing and a value) with an edge (the property) linking them. In the RDF metamodel, a pair of nodes and its edge are called a **triple**, because it consists of three parts (two nodes and one edge). The RDF metamodel is a directed graph, so it identifies one node (the one from which the edge is pointing) as the **subject** of the triple, and the other node

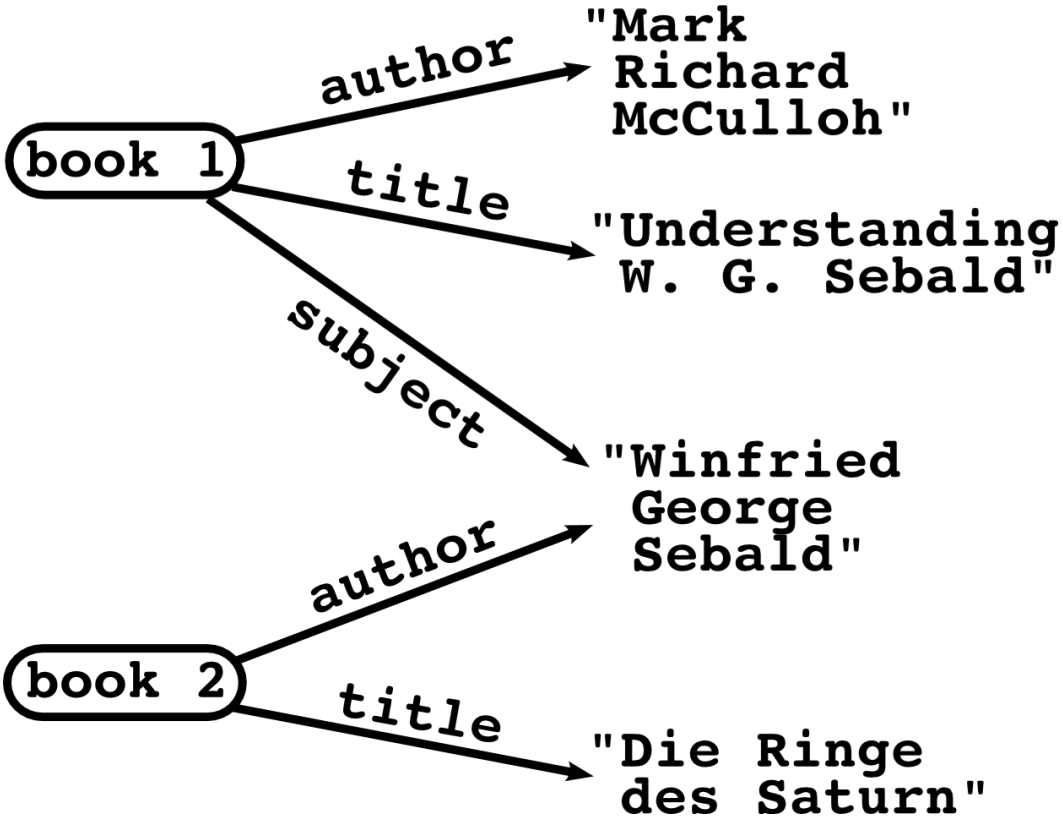

Figure 5. Treating properties as edges rather than nodes

(the one to which the edge is pointing) as its **object**. The edge is referred to as the **predicate** or *property* of the triple.

Figure 6 lists separately all the triples in Figure 5. There's something misleading about Figure 6, however. Figure 5 clearly indicates that the *Winfried George Sebald* who is the subject of book 1 is the same *Winfried George Sebald* who is the author of book 2. In Figure 6, this relationship is not clear. How can one tell if the *Winfried George Sebald* of the third triple is the same as the *Winfried George Sebald* of the triple statement? For that matter, how can one tell if the first three triples all involve the same book 1? This is easy to show in a diagram of the entire description graph, where there can be multiple edges attached to a node. But when that graph is disaggregated into triples, there must be some way of uniquely referring to nodes: identifiers. When two triples have nodes with the same identifier, it is clear that they are the same node. RDF achieves this by identifying nodes with Uniform Resource Identifiers (URIs).

The need to identify nodes when one breaks down an RDF graph into triples becomes important when "writing down" RDF graphs—creating textual representations of them instead of depicting them—so that they can be exchanged as data. Tree structures like XML do not necessarily have this problem, because it is possible to textually represent a tree structure without having to mention any node more than once. Thus, one price paid for the

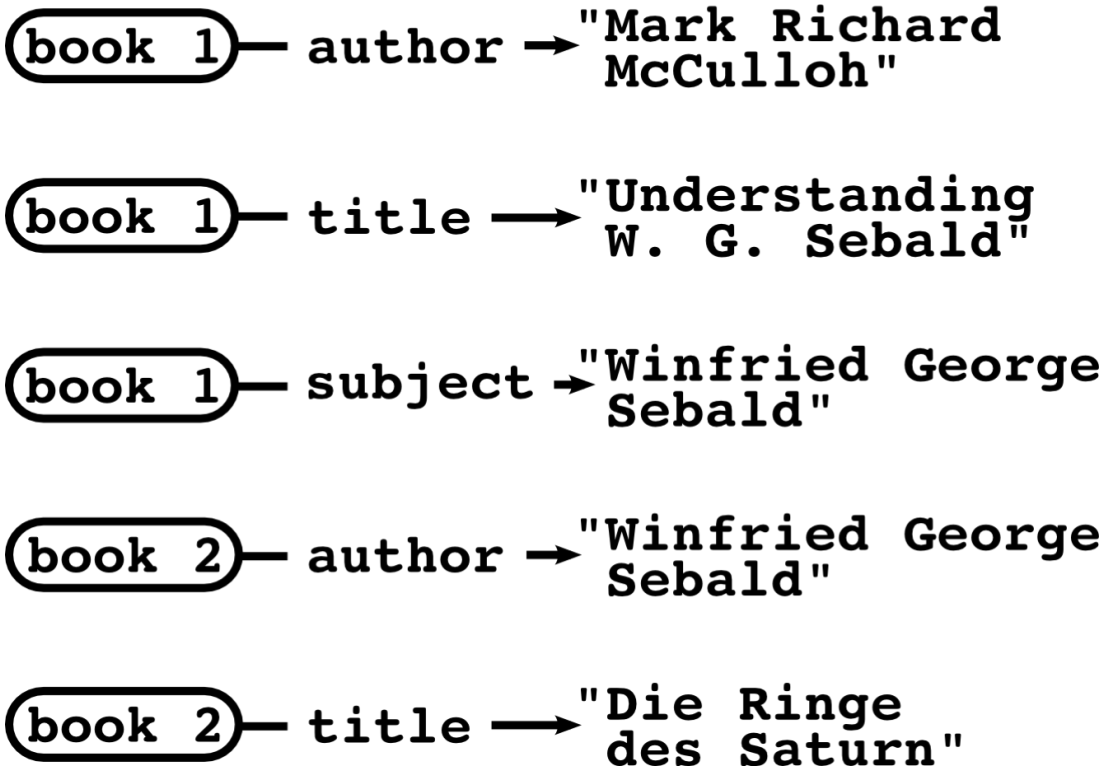

Figure 6. Listing triples individually.

generality and flexibility of graph structures is the added complexity of recording or writing those structures.

1.2.4 Choosing constraints

This tradeoff between flexibility and complexity illustrates a more general point about constraints. The word *constraint* has a negative connotation: who wants to be constrained? But in the context of managing and interacting with descriptions, constraints are a good thing. As discussed above, a tree is a graph with very specific constraints. These constraints allow you to do things with trees that aren't possible with graphs in general, such as represent them textually without repeating yourself, or uniquely identify nodes by the path from the root of the tree to that node. This can make managing descriptions and the things they describe easier and more efficient—*if* a tree structure is a good fit to the requirements of the organizing system. For example, an ordered tree structure like the Infoset is a good fit for the hierarchical structure of the content of a book. On the other hand, the network of relationships among the people and organizations that collaborated to produce a book might be better represented using a graph structure like RDF.

1.3 Modeling within constraints

A metamodel imposes certain constraints on the structure of descriptions. But in organizing systems, it is usually necessary to further specify the structure and composition of descriptions of the specific types of things being organized. For example, when designing a system for organizing books, it is not enough to say that a book's description is structured using XML; one needs to specify further that a book description includes a list of contributors, each entry of which provides a name and indicates the role of that contributor. This kind of specification is a *model* to which descriptions of books are expected to conform.

When designing an organizing system one may choose to reuse a standard model. For example, ONIX for Books is a standard model (conforming to the XML metamodel) developed by the publishing industry for describing books. If no such standard exists, or existing standards do not suit one's needs, a new model may be created for the specific domain. But typically one will not create a new metamodel: instead a choice is made from among the metamodels such as JSON, XML, or RDF that have been formally recognized and incorporated into existing standards.

1.3.1 Specifying vocabularies and schemas

Creating a model for descriptions of things in a particular domain involves specifying the common elements of those descriptions, and giving those elements standard names. The model may also specify how these elements are arranged into larger structures, for example

how they are ordered into lists nested into tree. Metamodels vary in the tools they provide for specifying the structure and composition of domain-specific models.

In XML, models are defined in separate documents known as **schemas**. An XML schema defining a domain model provides a vocabulary of terms that can be used as element and attribute names in XML documents that adhere to that model. For example, the Onix for Books schema specifies that an author of a book should be called a `Contributor`, and that the page count should be called an `Extent`. An XML schema also define rules for how those elements, attributes, and their content can be arranged into higher-level structures. For example, the Onix for Books specifies that the description of a book must include a list of `Contributor` elements, that this list must have at least one element in it, and that each `Contributor` element must have a `ContributorRole` child element.

If an XML schema is given an identifier, XML documents can use that identifier to indicate that they use terms and rules from that schema. An XML document may use vocabularies from more than one XML schema. XML also provides some tools for indicating which vocabularies are being used and for **validation**: automatically checking that vocabulary terms are being used correctly. (What exactly "correctly" means depends on the specific tools being used, as different tools vary in what they can validate.)

If two descriptions share the same XML schema and use only that schema, then combining them is straightforward. If not, it can be problematic, unless someone has figured out exactly how the two schemas should "map" to one another. Finding such a mapping is not a trivial problem, as XML schemas may differ semantically, lexically, structurally, or architecturally despite sharing a common implementation.

Because tree structures can vary considerably while still conforming to the XML Infoset metamodel, XML schemas must specify rules such as which elements can be children or parents of which other elements, or how many children a particular element may have. This is not necessary with RDF, because graphs that conform to the RDF metamodel all have the same structure: they are all sets of triples. This shared structure makes it simple to combine different RDF descriptions. Again, however, this simplicity comes at a cost: it is easy to combine RDF descriptions because puts fewer constraints on the structure of descriptions. This lack of constraints means that, in contrast to XML, it is not easy to check whether an RDF description is "correct."

Because structure is already defined by the RDF metamodel, defining a domain-specific model in RDF mainly involves specifying URIs and names for predicates. A set of RDF predicate names and URIs is known as an **RDF vocabulary**. Publication of vocabularies on the Web and the use of URIs to identify and refer to predicate definitions are key principles of Linked Data and knowledge graphs.

For example, the Resource Description and Access (RDA) standard for cataloging library resources includes a set of RDF vocabularies defining predicates usable in cataloging descriptions. One such predicate is `<http://rdvocab.info/Elements/extentOfText>` which is defined as "the number and type of units and/or subunits making up a resource consisting of text, with or without accompanying illustrations." The vocabulary further specifies that this predicate is a refinement of a more general predicate `<http://rdvocab.info/Elements/extent>` which can be used to indicate "the number and type of units and/or subunits making up a resource" regardless of whether it is textual or not.

RDF and XML each provide different, metamodel-specific tools to define a model for a specific domain. But not every metamodel provides such tools. JSON, for example, lacks any standardized way to define which terms can be used. That does not mean that one cannot use a standard vocabulary when creating descriptions using JSON, only that there is no agreed-upon way to use JSON to communicate which vocabulary is being used, and no way to automatically check that it is being used correctly.

1.3.2 Controlling values

So far, I've focused on how models specify vocabularies of terms and how those terms can be used in descriptions. But models may also constrain the values or content of descriptions. Sometimes, a single model will define both the terms that can be used for property names and the terms that can be used for property values. For example, some XML schemas may define the elements (properties) of a description and enumerate valid content (values) for those elements.

Often, however, there are separate, specialized vocabularies of terms intended for use as property values in descriptions. Typically these vocabularies provide values for use within statements that describe what an information object is about. Examples of such subject vocabularies include the Library of Congress Subject Headings (LCSH) and the Medical Subject Headings (MeSH). Other vocabularies (naming authorities) may provide preferred names for people, corporations, or places. Classification schemes are yet another kind of vocabulary, providing values for use in descriptive statements that classify things.

Because different metamodels such as XML and RDF take different approaches to specifying vocabularies, there will usually be different versions of these vocabularies for use with different metamodels. For example the LCSH are available both as XML conforming to the MADS (Metadata Authority Description Standard) schema, and as RDF using the SKOS (Simple Knowledge Organization System) vocabulary.

Specifying a vocabulary is just one way models can control what values can be assigned to properties. Another strategy is to specify what *types* of values can be assigned. For example, a

model for book descriptions may specify that the value of a `pages` property must be a number. Or it could be more specific and specify that the value must be a positive integer. Specifying a type like this narrows down the set of possible values for the property without necessarily having to enumerate every possible value.

In addition to or in lieu of specifying a type, a model may specify an encoding scheme for values. An **encoding scheme** is a specialized writing system or syntax for particular types of values. For example, Atom is an XML model for describing syndicated Web content such as blogs. It defines a published element for stating when a piece of content (such as a blog post) was published. Atom specifies that the content of the published element must be a date. But there are many different ways to write dates: `9/2/76`, `2 Sept. 1976`, `September 2nd 1976`, etc. So, Atom also specifies an encoding scheme for date values. The encoding scheme is RFC3339, a standard for writing dates. When using RFC3339, one always writes a date using the same form: `1976-09-02`.

Encoding schemes are often defined in conjunction with standardized identifiers. For example, International Standard Book Numbers (ISBNs) are not just sequences of Arabic numerals: they are values written using the ISBN encoding scheme. This scheme specifies how to separate the sequence of numerals into parts, and how each of these parts should be interpreted. The ISBN `978-3-8218-4448-0` has five parts, the first three of which indicate that the thing with this identifier is 1) a product of the book publishing industry, 2) published in a German-speaking country, and 3) published by the publishing house Eichborn.

Encoding schemes can be viewed as very specialized models of particular kinds of information, such as dates or book identifiers. But because they specify not only the structure of this information, but also how it should be written, they can be viewed as specialized writing systems. That is, encoding schemes specify how to *textually represent* information.

The second half of this article focuses on the issues involved in textually representing descriptions—writing them down. Graphs, trees, dictionaries, lists, and sets are general types of structures found in different metamodels. Thinking about these broad types and how they fit or don't fit the ways one wants to model descriptions can help with selecting a specific metamodel. Specific metamodels such as the Infoset or RDF are formalized and standardized definitions of the more general types of structures discussed above. Once a metamodel has been selected, one knows the high-level constraints one has to work with when modeling things. But because metamodels are abstract and exist only on a conceptual level, they can only do so much. To create, store, and exchange individual descriptions, the structures defined by an abstract metamodel need to be made concrete. They need to be written down.

The title is *Die Ringe des Saturn* and it has 371 pages.

```
{"title":"Die Ringe des Saturn","pages":371}
```

```
<book>
  <title>Die Ringe des Saturn</title>
  <pages>371</pages>
</book>
```

```
<http://lccn.loc.gov/96103072>
  <http://rdvocab.info/Elements/title> "Die Ringe des Saturn"@de
  ;
  <http://rdvocab.info/Elements/extentOfText> “371 p.”
  .
```

Table 1. Different ways of writing part of a book description.

2 Writing descriptions

Suppose that I am organizing books, and I have decided that it is important for the purposes of this organizing to know the title of each book and how many pages it has. Before me I have a book, which I examine to determine that its title is *Die Ringe des Saturn* and it has 371 pages. How should I write this down? There are an infinite number of forms I might choose. Table 1 lists a few. Let's examine these various forms of writing to see what they have in common and where they differ.

2.1 Notations

First, let's look at the actual marks on the page. To write you must make marks or—more likely—select from a menu of marks using a keyboard. In either case, you are using a **notation**: a set of characters with distinct forms.[5] The Latin alphabet is a notation, as are Arabic numerals. Some more exotic notations include alchemical symbols[6] and the symbols used for editorial markup. The characters in a notation usually have an ordering. Arabic numerals are ordered *1 2 3* ... American children usually learn the ordering of the Latin alphabet in the form of a song.

[5] The terminology here and in the following sections is borrowed from Roy Harris, *Signs of Writing*, 1995.

[6] See http://unicode.org/charts/PDF/U1F700.pdf.

A character may belong to more than one notation. The examples in Table 1 use characters from a few different notations: the letters of the Latin alphabet, Arabic numerals, and a handful of auxiliary marks: `. { } " : < > / $` Collectively, all of these characters—alphabet, numerals, and auxiliary marks—also belong to a notation called the American Standard Code for Information Interchange, or ASCII.

ASCII is an example of a notation that has been codified and standardized for use in a digital environment. A traditional notation like the Latin alphabet can withstand a certain degree of variation in the form of a particular mark. Two people might write the letter *A* rather differently, but as long as they can mutually recognize each other's marks as an "A", they can successfully share a notation. Computers, however, can't easily accommodate such variation. Each character must be strictly defined. In the case of ASCII, each character is given a number from 0 to 127, so that there are 128 ASCII characters.[7] When using a computer to type ASCII characters, each key you press selects a character from this "menu" of 128 characters. A notation that has had numbers assigned to its characters like this is called a **character encoding**.

The most ambitious character coding in existence is Unicode, which as of version 6.0 assigns numbers to 109,449 characters. Unicode makes the important distinction between **characters** and **glyphs**. A character is the smallest meaningful unit of a written language. In alphabet-based languages like English, characters are letters; in languages like Chinese, characters are ideographs. Unicode treats all of these characters as abstract ideas (*Latin capital A*) rather than specific marks (A *A* A). A specific mark that can be used to depict a character is a glyph. A *font* is a collection of glyphs used to depict some set of characters. A Unicode font explicitly associates each glyph with a particular number in the Unicode character encoding. The inability of computers to use contextual understanding to bridge the gap between various glyphs the abstract character depicted by those glyphs turns out to have important consequences for organizing systems.

Different notations may include very similar marks. For example, modern music notation includes marks for indicating the pitch of note, known as accidentals. One of these marks is ♯ ("sharp"). The sharp sign looks very much like the symbol used in American English as an abbreviation for the word *number*, as in *We're #1!* If you were to write a sharp sign and a number sign by hand, they would probably look identical. In a non-digital environment, one would rely on context to understand whether the written mark was being used as part of music notation, or mathematical notation, or as an English abbreviation.

---

[7] Only 95 of these characters are actually "marks" in the sense of being visible and printable. The other 33 are "control codes" that indicate things like the ends of printed lines. One can think of these as special auxiliary marks similar to the kind of symbols editors and proofreaders use to annotate texts.

Computers, however, have no such intuitive understanding of context. Unicode encodes the number sign and the sharp sign as two different characters. As far as a computer using Unicode is concerned, ♯ and # are completely different, and the fact that they have similar-looking glyphs is irrelevant. That's a problem if, for example, a cataloger has carefully described a piece of music by correctly using the sharp sign, but a person looking for that piece of music searches for descriptions using the number sign (since that's what you get when you press the keyboard button with the symbol that most closely resembles a sharp sign).

2.2 Writing systems

A **writing system** employs one or more notations, and adds a set of rules for using them. Most writing systems assume knowledge of a particular human language. These writing systems are known as *glottic* writing systems. But there are many writing systems, such as mathematical and musical ones, that are not tied to human languages in this way. Many of the writing systems used for describing things belong to this latter group, meaning that (at least in principle) they can be used with equal facility by speakers of any language.

Glottic writing systems, being grounded in natural human languages, are difficult to describe precisely and comprehensively. Non-glottic writing systems, on the other hand, can be described precisely and comprehensively using an abstract model. That is the connection between the structural perspective taken in the previous section, and the textual perspective taken in this section. A non-glottic writing system is described by a particular metamodel, and structures that fit within the constraints of a given metamodel can be textually represented using one or more writing systems that are described by that metamodel.

Some writing systems are closely identified with specific metamodels. For example, XML and JSON are *both* 1) metamodels for structuring information *and* 2) writing systems for textually representing information. In other words, they specify both the abstract structure of a description and how to write it down. It's possible to conceive of other ways to textually represent the structure of these metamodels, but for each of these metamodels just one writing system has been standardized.[8]

RDF, on the other hand, is *only* a metamodel, not a writing system. RDF only defines an abstract structure, not how to write that structure down. So how does one write down information that is structured as RDF? It turns out that there are many choices. Unlike XML and JSON, several different writing systems for the RDF metamodel have been

[8] An alternative writing system for XML-structured data has been standardized: EXI (Efficient XML Interchange). However it is not yet widely used.

standardized, including N-Triples, Turtle, RDFa, JSON-LD, and RDF/XML.[9] Each of these is a writing system that is abstractly described by the RDF metamodel.

Writing systems provide rules for arranging characters from a notation into meaningful structures. A character in a notation has no inherent meaning. Characters in a notation only take on meaning in the context of a writing system that uses that notation. For example: what does the letter *I* from the Latin alphabet mean? That question can only be answered by looking at how it is being used in a particular writing system. If the writing system is American English, then whether *I* has a meaning depends on whether it is grouped with other letters or whether it stands alone. Only in the latter case does it have an assignable meaning. However in the arithmetic writing system of Ancient Rome, which also uses as a notation the letters of the Latin alphabet, *I* has a different meaning: *one*.

This example also serves to illustrate how the ordering of a notation can differ from the ordering of a writing system that uses that notation. According to the ordering of the Latin alphabet, the twelfth letter *L* comes before the twenty-second letter *V*. But in the Roman arithmetic writing system, *V* (the number 5) comes before *L* (the number 50). Unless one knows which ordering is being used, the letters *L* and *V* cannot be arranged "in order."

This kind of difference in ordering can arise in more subtle ways as well. When names are sorted "alphabetically," first the first characters of each name are compared and arranged according to the ordering of the writing system.[10] If the first characters of two names are the same, then the second characters are compared, and so on. This same kind of ordering procedure can be applied to sequences of numerals. Then *334* will come before *67*, because *3* (the first character of the first sequence) comes before *6* (the first character of the second sequence) according to the ordering of the notation (Arabic numerals). However, it is more common when ordering sequences of numerals to treat them as decimal numbers, and thus to use the ordering imposed by the decimal system. In the decimal writing system, *67* comes before *334*, since the latter is a bigger number.

This difference is important for organizing systems. Computers will sort values differently depending on whether they are treating sequences of numerals as numbers or just as sequences. Some organizing systems mix multiple ways of ordering the same characters. For example, Library of Congress call numbers have four parts, and sequences of Arabic numerals can appear in three of them. In the second part, indicating a narrow subject area,

---

[9] Confusingly, the last of these is a writing system that uses XML syntax to textually represent RDF structure. This means that while XML tools can read and write RDF/XML, they cannot manipulate the graph structures it represents, because they were designed to work with XML's tree structures. Similarly, JSON-LD is a writing system that uses JSON syntax to textually represent RDF structure, with similar advantages and drawbacks to RDF/XML.

[10] Alphabetization relies on the ordering of the writing system, not the notation. For example, Swedish and German are two writing systems that assign different orderings to the same notation.

and fourth part, indicating year of publication, sequences of numerals are treated as numbers and ordered according to the decimal system. In the third part, however, sequences of numerals are treated as sequences and ordered "notationally" as in the example above (*334* before *67*).

Differences in ordering demonstrate just one way that multiple writing systems may use the same notation differently. For example, the American English and British English writing systems both use the same Latin alphabet, but impose slightly different spelling rules. The Japanese writing system employs a number of notations, including traditional Chinese characters (***kanji***) as well as the Latin alphabet (***rōmaji***). Often writing systems do not share the same exact notation but have mostly overlapping notations. Many European languages, for example, extend the Latin alphabet with characters such as *Å* and *Ü* that add additional marks, known as diacritics, to the basic characters.[11]

Often in organizing systems, it is necessary to represent values from one writing system in another writing system that uses a different notation, a process known as **transliteration**. For example, early computer systems only supported the ASCII notation, so text from writing systems that extend the Latin alphabet had to be converted to ASCII, usually by removing (or sometimes transliterating) diacritics. This made the non-ASCII text usable in an ASCII-based computerized organizing system, at the expense of information loss.

Even in modern computer systems that support Unicode, however, transliteration is often needed to support organizing activities by users who cannot read text written using its original system. The Library of Congress and the American Library Association provide standard procedures for transliterating text from over sixty different writing systems into the (extended) Latin alphabet.

2.3 Syntax

The examples in Table 1 express the same information using five different writing systems. The five examples use the same notation (ASCII) but differ in their **syntax**: the rules that define how characters can be combined into words and how words can be combined into higher-level structures.[12]

---

[11] Since ASCII cannot represent these characters, a whole family of character encodings was created, ISO- 8859, in which each encoding enumerates 256 instead of ASCII's 128 characters. Each encoding thus has more space to accommodate the additional characters of regionally-specific notations. ISO 8859-5, for example, has extensions to support the Cyrillic alphabet.

[12] In discussions of glottic writing systems, "syntax" usually refers only to the rules for combining words into sentences. In discussions of programming languages, "syntax" has the broader sense that I use here.

Consider the first example: *The title is* Die Ringe des Saturn *and it has 371 pages*. The period ending this sequence of characters indicates to us that this is a sentence. This sentence is one way the American English writing system might be used to express two statements about the book being described. A *statement* is one distinct fact or piece of information. In glottic writing systems like American English, there is usually more than one sentence one could write to express the same statement. For example, instead of *it has 371 pages* one might have written *the number of pages is 371*. American English writing also enables the construction of complex sentences that express more than one statement.

In contrast, when creating descriptions of things in an organizing system one generally uses non-glottic writing systems in which each sentence only expresses a single statement, and there is just one way to write a sentence that expresses a given statement.[13] These restrictions make these writing systems less expressive, but simplify their use. In particular, since there is a one-to-one correspondence between sentences and statements, one can drop the distinction and just talk about the statements of a description.

Now consider the structure of the statement, *The title is* Die Ringe des Saturn *and it has 371 pages*. Spaces are used to separate the text into words, and American English syntax defines the functions of those words. The verb is in this statement functions to link the word title to the phrase *Die Ringe des Saturn*. This is typical of the kind of statements found in a description. Each statement identifies and describes some aspect of the thing. In this case, the statement assigns the value *Die Ringe des Saturn* to the property *title*.

Descriptions can be analyzed as involving properties or attributes of things and their corresponding values or content. In a writing system like American English, it is not always so straightforward to determine which words refer to properties and which refer to values. (This is one reason why blobs of text are not ideal description structures.) Writing systems designed for expressing descriptions, on the other hand, usually define syntax that makes this determination easier. In the dictionary examples above, an arrow character → was used to separate properties and values.

This ease of distinguishing properties and values comes at a price, however. The syntax of English is forgiving: one can read a sentence with somewhat garbled syntax such as *371 pages it has* and often still make out its meaning.[14] This is usually not the case with writing systems intended for expressing descriptions. These systems strictly define their rules for how

---

[13] In truth, even non-glottic writing systems designed to encode descriptions unambiguously can have variant forms of the same statement. For example, XML permits some variation in the way the same Infoset may be textually represented. Often these variations involve the treatment of content that may under some circumstances be treated as optional, such as whitespace. The difference is that in writing systems designed for description, these variations can be precisely enumerated and rules developed to reconcile them, while this is not generally true for glottic writing systems.

[14] Fortunately for Yoda.

characters can be combined into higher-level structures. Structures that follow the rules are *well formed* according to that system.

Take for example the second entry in Table 1. This example is written in JSON. As explained earlier, JSON is a metamodel for structuring information using lists and dictionaries. But JSON is also a writing system, which borrows its syntax from JavaScript. The JSON syntax uses brackets to textually represent lists `[1,2,3]` and braces to textually represent dictionaries `{"Die Ringe des Saturn":"Austerlitz", "pages":371}`. Within braces, the colon character `:` is used to link properties with their values, much as is was used in the previous example. So `"pages":371` is a statement assigning the value `371` to the property `pages`.

The third example in Table 1 is written in XML. Like JSON, XML is a metamodel and also a writing system. Here, instead of having a value assigned to a property in a JSON dictionary, an XML element is used. XML elements are textually represented as *tags* that are marked using the special characters <, > and /. So, this fragment of XML consists of a `book` element with two child elements, `title` and `pages`, each of which has some text content.

The fourth example in Table 1 is a fragment of Turtle, one of the writing systems for RDF. Turtle provides a syntax for writing down RDF triples. Each triple consists of a subject, predicate, and object separated by spaces. Recall that RDF uses URIs to identify subjects, predicates, and some objects; these URIs are written in Turtle by enclosing them in angle brackets `< >`. Triples are separated by period `.` characters, but triples that share the same subject can be written more compactly by writing the subject only once, and then writing the predicate and object of each triple, separated by a semicolon `;` character. This is what is shown in Table 1: two triples that share a subject.

3 Worlds of description

Descriptions are both designed objects with particular structures, and written documents with particular syntaxes. There are many possible choices of structure and syntax. But these choices are never made in isolation. Just as an architect or designer must work within the constraints of the existing built environment, and just as any author must work with existing writing systems, descriptions are always created as part of a pre-existing "world" over which any one of us has little control.

Choices of structure and syntax have converged historically into broad patterns of usage. For lack of a better term, I will call these broad patterns "worlds." "World" is not a technical term and shouldn't be taken too literally: the broad areas of application sketched here have considerable overlap, and there are many other ways one might identify patterns of description structure and syntax. That said, the three "worlds" described here do reflect real patterns of description form that influence tool and technology choices. In your own work

creating and managing descriptions, it is likely that you will need to think about how your descriptions fit into one or more of these worlds.

3.1 The document processing world

The first world is concerned primarily with the creation, processing and management of hybrid narrative-transactional documents such as instruction manuals, invoices, textbooks, or annotated medieval manuscripts. These are quite different kinds of documents, but they all contain a mixture of narrative text and structured data, and they all can be usefully modeled as tree structures. Because of these shared qualities, tools as different as publishing software, supply-chain management software, and scholarly editing software have all converged on common XML-based solutions. ("The XML world" would be another appropriate name for the document processing world.)

This convergence was no accident, because XML was designed specifically to address the problem of how to add structure and data to narrative documents by "marking them up." XML is the descendant of SGML (Standard Generalized Markup Language), which in turn descended from IBM's Generalized Markup Language, which was invented to enable the production and management of large-scale technical documentation.

The abstract data model underlying XML is called the XML Information Set or Infoset. The Infoset defines a document as a partially ordered tree of parts or "information items." Every XML document can thus be understood as a tree (although because the Infoset defines an XML document as a very specific kind of tree, not every tree structure is expressible as an XML document). Thus tools and technologies in the document processing world are built for manipulating and combining tree structures.

A "toolchain" is set of tools intended to be used together to achieve some goal. The XML toolchain is quite comprehensive. It consists of tools for creating XML documents (XML editors), tools for expressing logical document and data models (XML Schema, RELAX NG, Schematron), tools for transforming XML documents (XSLT), tools for describing document processing "pipelines" (XProc), and tools for storing and querying collections of XML documents (XML databases, queried using XQuery). Used together, these tools provide very powerful means of working with tree-structured documents.

Most programming languages also provide libraries for working with XML. This fact has led some to believe that XML is a kind of "universal" format for exchanging data among systems. This is a misconception. The XML Infoset does not map easily to the data structures commonly found in most programming languages. "Working with XML" in most programming languages means translating from the native tree structures of XML to data structures native to that language, usually meaning lists and dictionaries. This translation can be problematic and often means giving up many of the strengths of XML.

But just because XML is not a universal solution for every possible problem does not mean that it isn't the best solution for a wide variety of problems. To gauge whether your descriptions are or ought to be part of the document processing world, ask yourself the following questions:

- Do my descriptions contain mixtures of narrative text and structured data?
- Can my descriptions easily be modeled as tree structures?
- Are the vocabularies I need or want to use made available primarily using XML technologies such as XML Schema?
- Do I need to work with a body of existing descriptions already encoded as XML?
- Do I need to interoperate with processes or partners that utilize the XML toolchain?

If the answer to one or more of these questions is "yes," then chances are good that you are working within the document processing world, and you will need to become familiar with conceptualizing your descriptions as trees and working with them using XML tools.

3.2 The Web world

The second "world" emerged in the early 1990s with the creation of the World Wide Web. In contrast to the document processing world, which developed in response to the need for management of large, complex publishing projects, the Web was developed to address a need for simple and rapid sharing of scientific data. Of course, it has grown far beyond that initial use case, and is now a ubiquitous infrastructure for all varieties of information and communication services.

Documents, data, and services on the Web are conceptualized as *resources*, identified using URIs, and accessible through *representations* transferred via the Hypertext Transfer Protocol (HTTP). Representations are sequences of bytes, and could be HTML pages, JPEG images, tabular data, or practically anything else transferrable via HTTP. No matter what they are, representations transferred over the Web include descriptions of themselves. These descriptions take the form of property-value pairs, known as *HTTP headers*. The HTTP headers of Web representations are structured as dictionaries.

Dictionary structures appear many other places in Web infrastructure. URIs may include a *query* component beginning with a `?` character. This component is used for purposes such as providing query parameters to search services. The query component is commonly structured as a dictionary, consisting of a series of property-value pairs separated by the `&` character. For example, the URI `https://www.google.com/search?q=sebald&tbs=qdr:m` includes the query component `q=sebald&tbs=qdr:m`. This is

a dictionary with the properties `q` and `tbs`, respectively specifying the search term and temporal constraints on the search.

Data entered into an HTML form is also structured as a dictionary. When an HTML form is submitted, the entered data is used either to compose the query component of a URI, or to create a new representation to be transferred to a Web server. In either case, the data is structured as a set of properties and their corresponding values.

HTML documents are structured as trees, but descriptions embedded within HTML documents can also be structured as dictionaries. HTML documents may include a dictionary of metadata elements, each of which specifies a property and its value. Recently support for *microdata* was added to HTML, which is another method of adding dictionaries of property-value pairs to documents. Using microdata, authors can annotate Web content with additional information, making it easier to automatically extract structured descriptions of that content. *Microformats* are another method for doing this by mapping existing HTML attributes and values to (nested) dictionary structures.

Dictionary structures are easy to work with in any programming language, and they pervade various popular frameworks for programming the Web. In the programming languages used to implement Web services, HTTP headers and query parameters are easily mapped to dictionary data structures native to those languages. And on the client side, there is only one programming language that runs within all Web browsers: JavaScript. The dictionary is the fundamental data structure within JavaScript as well.

Thus it is unsurprising that JSON, a dictionary-structured, JavaScript-based syntax, has become the de facto standard for exchanging data on the web. Web services providing structured data intended for programmatic use can make that data available as JSON, which is well-suited for use either by JavaScript programs running within browsers, or by programs written in other languages running outside of browsers (for example, smartphone applications).

It's worth noting that this convergence on dictionary structures for Web data was not seen as inevitable. For many years, it was assumed that the Web would be part of the document-processing world described above. The World Wide Web consortium pursued a vision of the Web as primarily XML-based, and sought to unify HTML and other Web technologies with the XML toolchain. It is now commonly accepted that this vision will not come to fruition, and that there are useful differences of approach between the document- processing world and the Web world.

This does not mean that the two worlds do not have significant overlaps. Some very important Web representation types are XML-based, such as the Atom syndication format. Trees will continue to be the structure of choice for Web representations that consist primarily of narrative rather than transactional data (although it is worth noting that the

dominant tree-structured representation format on the Web is HTML, not XML). But for structured descriptions intended to be accessed and manipulated on the Web, dictionary structures currently rule. To gauge whether your descriptions are or ought to be part of the Web world, ask yourself the following questions:

- Is the Web the primary platform upon which I will be making my descriptions available?
- Are my descriptions primarily structured, transactionally-oriented data?
- Can my descriptions easily be modeled as lists of properties and values (dictionaries)?
- Are the vocabularies I need or want to use made available primarily using HTML technologies such as microdata or microformats?
- Do I need to make my descriptions easily usable from within a wide array of programming languages?

If the answer to one or more of these questions is "yes," then chances are good that you are working within the Web world, and you will need to become familiar with conceptualizing your descriptions as dictionaries and working with them using programming languages such as JavaScript.

3.3 The Linked Data world

The vision of a Linked Data world builds upon the Web world, but adds some further prescriptions and constraints for how to structure descriptions. The Linked Data world unifies the concept of a description with the Web notion of a resource as anything with a URI. In Linked Data, anything being described must have a URI. Furthermore, the descriptions must be structured as graphs, adhering to the RDF metamodel and relating resources to one another via their URIs. Advocates of Linked Data further prescribe that those descriptions must be made available as representations transferred over HTTP.

This is a departure from the Web world. The Web world is also structured around URIs, but it does not require that every resource being described have a URI. For example, in the Web world a list of bibliographic descriptions of books by W. G. Sebald might be published at a specific URI, but the individual books themselves might not have URIs. In the Linked Data world, in addition to the list having a URIs, each book would have a URI too, in addition to whatever other identifiers it might have.

Making an HTTP request to an individual book URI would then return a graph-structured description of that book. This, too, is a departure from the Web world, which is agnostic about the form representations or descriptions of resources should take (although dictionary structures are often favored on the Web when the clients consuming those descriptions are computer programs). In Linked Data, all descriptions are structured as

RDF graphs. Each description graph links to other description graphs by referring to related resources using their URIs, so that, in theory, all description graphs are linked into a single massive graph structure. In practice, however, it is far from clear that this is an achievable, or even a desirable, goal.

A significant number of descriptions have already been made available in accordance with the principles outlined above. Descriptions published according to these principles are often referred to as "knowledge graphs." Prominent examples include Wikidata, a graph of descriptions of things mentioned in Wikipedia articles, the Virtual International Authority File (VIAF), a graph of descriptions of names collected from various national libraries' name authority files, GeoNames, a graph of descriptions of places, and Europeana, a graph of descriptions of cultural heritage artifacts.

Having a common way of identifying resources (the URI) and a single shared metamodel (RDF) for all descriptions makes it much easier to combine descriptions from different sources. To gauge whether your descriptions are or ought to be part of the Semantic Web world, ask yourself the following questions:

- Is the Web the primary platform upon which I will be making my descriptions available?
- Is it important that I be able to easily and freely aggregate the elements of my descriptions in different ways and to combine them with descriptions created by others?
- Are my descriptions best modeled as graph structures?
- Have the vocabularies I need or want to use been created using RDF?
- Do I need to work with a body of existing descriptions that have been published as Linked Data?

If the answer to one or more of these questions is "yes," then chances are good that you should be working within the Semantic Web world, and you ought to become familiar with conceptualizing your descriptions as graphs and working with them using Semantic Web tools.