Chapter 11

Semistructured Data Model
Relations are structured data

All tuples have the same structure (same set of attributes).
A text (and most other data) are unstructured data

Despite federal safety investigations of Tesla’s self-driving cars, the company’s chief executive, Elon Musk, is hardly backing off on his grand plans for autonomous vehicles.

In a blog post late Wednesday, Mr. Musk updated Tesla’s “master plan” with a pledge to expand beyond electric cars into battery-powered pickups, semitrucks and buses, and to equip them with advanced self-driving systems.

He made no mention of the fatal May 7 accident involving a Tesla Model S with its Autopilot system engaged, or of the federal scrutiny of the technology. A criticism of that system has been that, despite its name, its collision-avoidance abilities depend on the human driver’s being ready to immediately retake control of the vehicle in a crisis.
If we describe data (in a structured way) along with its structure, then it is semi-structured data.
11.1.2 Semistructured Data Representation: Example
11.1.2 Semistructured Data Representation: Example
A database of **semistructured data** is a collection of **nodes**.

Each node is either a **leaf** or **interior**:

- **Leaf nodes** have associated **data**. The type of this data can be any atomic type, such as numbers and strings.

- **Interior nodes** have one or more arcs out. **Each arc has a label**, which indicates how the node at the head of the arc relates to the node at the tail.

- One **interior node**, called the **root**, has no arcs entering and represents the **entire database**.

- Every node must be reachable from the root, although the graph structure is not necessarily a tree.
A label $L$ on the arc from node $N$ to node $M$ can play one of two roles:

1. It may be possible to think of $N$ as representing an object or entity, while $M$ represents one of its attributes. Then, $L$ represents the name of the attribute.

2. We may be able to think of $N$ and $M$ as objects or entities and $L$ as the name of a relationship from $N$ to $M$.
XML: Extensible Markup Language
XML is a tag-based notation designed originally for “marking” documents, much like the familiar HTML.

While HTML’s tags talk about the representation of the information contained in documents (for instance, which portion is to be displayed in italics or what the entries of a list are), XML tags are intended to talk about the meanings of pieces of the document.

In this section, we shall introduce the basics of XML.

- We shall see that it captures, in a linear form, the same structure as do the graphs of semistructured data introduced earlier.

- In particular, tags can play the same role as the labels on the arcs of a semistructured-data graph.
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
  </Star>
  <Star>
    <Name>Mark Hamill</Name>
    <Address>
      <Street>456 Oak Rd.</Street>
      <City>Brentwood</City>
    </Address>
  </Star>
  <Movie>
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>
11.2.1 Semantic Tags

Tags in XML are text surrounded by *triangular brackets*, i.e., `< · · · >`, as in HTML.

Also as in HTML, *tags generally come in matching pairs*:

- with an **opening tag** like `<Foo>` and a matched **closing tag** that is the same word with a slash, like `< /Foo>`.

- Between a matching pair `<Foo>` and `< /Foo>`, there can be text, including text with nested HTML tags, and any number of other nested matching pairs of XML tags.

A pair of matching tags and everything that comes between them is called an **element**.
A single tag, with no matched closing tag, is also permitted in XML.

- In this form, the tag has a slash before the right bracket, for example, `<Foo/>`.

- Such a tag cannot have any other elements or text nested within it. It can, however, have attributes.
XML is designed to be used in two somewhat different modes:

1. Well-formed XML.
2. Valid XML.
11.2.2 XML With and Without a Schema

Well-formed XML:

- Well-formed XML allows you to invent your own tags, much like the arc-labels in semistructured data.

- This mode corresponds quite closely to semistructured data, in that there is no predefined schema, and each document is free to use whatever tags the author of the document wishes.

- Of course, the nesting rule for tags must be obeyed, or the document is not well-formed.
11.2.2 XML With and Without a Schema

Valid XML:

- Valid XML involves a “DTD” (Document Type Definition) that specifies the allowable tags and gives a grammar for how they may be nested.

- This form of XML is intermediate between the strict-schema model such as the relational model, and the completely schemaless world of semistructured data.

- As we shall see later, DTD’s generally allow more flexibility in the data than does a conventional schema. DTD’s often allow optional fields or missing fields, for instance.
The minimal requirement for well-formed XML is that:

- the document begin with a declaration that it is XML,
- and that it have a root element that is the entire body of the text.

Thus, a well-formed XML document would have an outer structure like:

```xml
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<SomeTag>
   ...
</SomeTag>
```
11.2.3 Well-Formed XML

```xml
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<SomeTag>
  ...
</SomeTag>
```

The first line indicates the file is an XML document.

The encoding UTF-8 (UTF = “Unicode Transformation Format”) is a common choice of encoding for characters in documents, because it is compatible with ASCII and uses only one byte for the ASCII characters.

The attribute `standalone = "yes"` indicates that there is no DTD for this document; i.e., it is well-formed XML.

Notice that the initial declaration is delineated by special markers `<? ... ?>`.

The root element for this document is labelled `<SomeTag>`. 
As in HTML, an XML element can have attributes (name-value pairs) within its opening tag.

An attribute is an alternative way to represent a leaf node of semistructured data.

Attributes, like tags, can represent labeled arcs in a semistructured-data graph.

Attributes can also be used to represent the “sideways” arc as in a previous figure.
In a previous example, we have such an element:

```xml
<Movie>
  <Title>Star Wars</Title>
  <Year>1977</Year>
</Movie>
```

The **title** or **year** children of the movie node could be represented directly in the `<Movie>` element, rather than being represented by nested elements. That is, we can represent the `<Movie>` element above by:

```xml
<Movie year=1977> <Title>Star Wars</Title></Movie>
```

We could even make both child nodes be attributes by:

```xml
<Movie title="Star Wars" year=1977></Movie>
```

Or even:

```xml
<Movie title="Star Wars" year=1977 />
```
11.2.5 Attributes That Connect Elements

An important use for attributes is to represent connections in a semistructured data graph that do not form a tree.

We shall see later how to declare certain attributes to be identifiers for their elements.

We shall also see how to declare that other attributes are references to these element identifiers.

For now, let us see an example of how these attributes could be used.
11.2.5 Attributes That Connect Elements: Example

```xml
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star StarID = "cf" starredIn = "sw">
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  <Star starID = "mh" starredIn = "sw">
    <Name>Mark Hamill</Name>
    <Address>
      <Street>456 Oak Rd.</Street>
      <City>Brentwood</City>
    </Address>
  </Star>
  <Movie movieID = "sw" startsOf = "cf", "mh">
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>
```
Namespace

Warren Buffett: A security guy?
11.2.6 Namespaces

There are situations in which XML data involves tags that come from two or more different sources, and which may therefore have conflicting names.

For example, we would not want to confuse an HTML tag used in text with an XML tag that represent the meaning of that text.

Later, we shall see how XML Schema requires tags from two separate vocabularies.

To distinguish among different vocabularies for tags in the same document, we can use a namespace for a set of tags.
11.2.6 Namespaces

To say that an element’s tag should be interpreted as part of a certain namespace, we can use the attribute `xmlns` in its opening tag.

There is a special form used for this attribute:

```
xmlns: name = "URI"
```

Within the element having this attribute, `name` can modify any tag to say the tag belongs to this namespace.

That is, we can create **qualified** names of the form `name:tag`, where `name` is the name of the namespace to which the tag `tag` belongs.
The **URI** (Universal Resource Identifier) is typically a URL referring to a document that describe the meaning of the tags in the namespace.

This description need not be formal. It could be an informal article about expectations.

In could even be nothing at all, and still serve the purpose of distinguishing different tags that had the same name.
Suppose that we want to say that in element **StarMovieData** shown previously, certain tags belong to the namespace defined in the document infolab.stanford.edu/movies.

We could choose a name such as **md** for the namespace by using the opening tag:

```
<md:StarMovieData xmlns:md="http://infolab.stanford.edu/movies" >
```

Our intent is that StarMovieData itself is part of this namespace, so it gets the prefix **md:**, as does its closing tag

```
</md:StarMovieData>
```

Inside this element, we have the option of asserting that tags of subelements belong to this namespace by prefixing their opening and tags with **md:**.
Information encoded in XML is not always to be stored in a database.

It has become common for computers to share data across the Internet by passing messages in the form of XML elements.

However, it is becoming increasingly common for XML to appear in roles traditionally reserved for relational databases.

People study efficient ways to store XML data in traditional databases. (We skip the details here.)
DTD: Document Type Definition
For a computer to process XML documents automatically, it is helpful for there to be something like a schema for the documents.

It is useful to know what kinds of elements can appear in a collection of documents and how elements can be nested.

The description of the schema is given by a grammar-like set of rules, called a document type definition, or DTD.

It is intended that companies or communities wishing to share data will each create a DTD that describes the forms of the data they share, thus establishing a shared view of the semantics of their elements.

For instance, there could be a DTD for describing protein structures, a DTD for describing the purchase and sale of auto parts, and so on.
The gross structure of a DTD is:

```xml
<!DOCTYPE root-tag [
  <!ELEMENT element-name (components)> 
  more elements 
]
```
<!DOCTYPE Stars [ 
 <!ELEMENT Stars (Star*)>  
 <!ELEMENT Star (Name, Address+, Movies)>  
 <!ELEMENT Name (#PCDATA)>  
 <!ELEMENT Address (Street, City)>  
 <!ELEMENT Street (#PCDATA)>  
 <!ELEMENT City (#PCDATA)>  
 <!ELEMENT Movies (Movie*)>  
 <!ELEMENT Movie (Title, Year)>  
 <!ELEMENT Title (#PCDATA)>  
 <!ELEMENT Year (#PCDATA)>  
 ]>
11.3.1 The Form of a DTD

The opening *root-tag* and its matched closing tag surround a document that conforms to the rules of this DTD.

*Element declarations, introduced by !ELEMENT*, give the tag used to surround the portion of the document that represents the element, and also give a parenthesized list of “components.”

The latter are elements that *may or must appear* in the element being described.

The exact requirement on components are indicated in a manner we shall see shortly.
There are two important special cases of components:

1. \texttt{(#PCDATA)} ("parsed character data").
2. The keyword \texttt{EMPTY}.
11.3.1 The Form of a DTD

1. (**PCDATA**) ("parsed character data"): 

   - (**PCDATA**) after an element name means that element has a value that is text, and it has no elements nested within.

   Parsed character data may be thought of as HTML text. It can have formatting information within it, and the special characters like `<` must be escaped, by `&lt;`, and similar HTML codes.

   For instance,

   ```
   <!ELEMENT Title (PCDATA) >
   ```

   says that between `<Title>` and `</Title>` tags, a character string can appear. However, any nested tags are not part of the XML; they could be HTML, for instance.
2. The keyword **EMPTY**:

- The keyword **EMPTY**, with no parentheses, indicated that the element is one of those that has no matched closing tag. It has no subelements, nor does it have text as a value. For instance,

  ```xml
  <!ELEMENT Foo EMPTY>
  ```

  says that the only way the tag **Foo** can appear is as `<Foo/>`. 
The components of an element $E$ are generally other elements. They must appear between the tags $<E>$ and $</E>$ in the order listed.

However, there are several operators that control the number of times elements appear:

1. A * following an element means that the element may occur any number of times, including zero times.

2. A + following an element means that the element may occur one or more times.

3. A ? following an element means that the element may occur either zero times or one time, but no more.
11.3.1 The Form of a DTD

- We can connect a list of options by the “or” symbol | to indicate that exactly one option appears. For example, if <Movie> elements has <Genre> subelements, we might declare these by

  ```xml
  <!ELEMENT Genre (Comedy|Drama|SciFi|Teen) >
  ```

to indicate that each <Genre> element has one of these four subelements.
We have shown this example of DTD:

<!DOCTYPE Stars [ 
  <!ELEMENT Stars (Star*)> 
  <!ELEMENT Star (Name, Address+, Movies)> 
  <!ELEMENT Name (#PCDATA)> 
  <!ELEMENT Address (Street, City)> 
  <!ELEMENT Street (#PCDATA)> 
  <!ELEMENT City (#PCDATA)> 
  <!ELEMENT Movies (Movie*)> 
  <!ELEMENT Movie (Title, Year)> 
  <!ELEMENT Title (#PCDATA)> 
  <!ELEMENT Year (#PCDATA)> 
]>
<Stars>

<Star>

<Name>Carrie Fisher</Name>

<Address>

<Street>123 Maple St.</Street>

<City>Hollywood</City>

</Address>

<Address>

<Street>5 Locust Ln.</Street>

<City>Malibu</City>

</Address>

</Star>

</Stars>

<Movies>

<Movie>

<Title>Star Wars</Title>

<Year>1977</Year>

</Movie>

<Movie>

<Title>Empire Strikes Back</Title>

<Year>1980</Year>

</Movie>

<Movie>

<Title>Return of the Jedi</Title>

<Year>1983</Year>

</Movie>

</Movies>
<Star>
  <Name>Mark Hamill</Name>
  <Address>
    <Street>456 Oak Rd.</Street>
    <City>Brentwood</City>
  </Address>
  <Movies>
    <Movie>
      <Title>Star Wars</Title>
      <Year>1977</Year>
    </Movie>
    <Movie>
      <Title>Empire Strikes Back</Title>
      <Year>1980</Year>
    </Movie>
    <Movie>
      <Title>Return of the Jedi</Title>
      <Year>1983</Year>
    </Movie>
  </Movies>
</Star>
<Star>
</Stars>
If a document is intended to conform to a certain DTD, we can either:

1. Include the DTD itself as a preamble to the document, or
2. In the opening line, refer to the DTD, which must be stored separately in the file system accessible to the application that is processing the document.

**Example:** Here is how we can assert that the previous document conforms to the previous DTD:

```xml
<?xml version = "1.0" encoding = "utf-8" standalone = "no" ?>
<!DOCTYPE Stars SYSTEM "star.dtd">
```
A DTD also lets us specify:

- which **attributes** an element may have, and
- what the **types** of these attributes are.

A declaration of the form

```
<!ATTLIST element-name attribute-name type>
```

says that the named attribute **can** be an attribute of the named element, and that the type of this attribute is the indicated type.

Several attributes can be defined in one ATTLIST statement, but it is not necessary to do so, and the ATTLIST statement can appear in any position in the DTD.
Common types for attributes include (but not limited to):

- **CDATA**. This type is essentially character-string.

- Enumerated type: A list of possible strings, surrounded by parentheses and separated by |’s.

Following the data type, there can be a keyword **#REQUIRED** or **#IMPLIED**, which means that the attribute must be present, or is optional, respectively.
11.3.3 Attribute Lists

Example: Instead of having the title and year be sub-elements of a <Movie> element, we could make these be attributes instead. Here is a possible example of attribute-list declarations:

```xml
<!ELEMENT Movie EMPTY>
<!ATTLIST Movie
    title CDATA #REQUIRED
    year CDATA #REQUIRED
    genre (comedy | drama | sciFi | teen) #IMPLIED
>
```

Notice that Movie is now an empty element. We have given it three attributes: title, year and genre. The first two are CDATA, while the genre has values from an enumerated type. Notice that in the document, the values, such as comedy, appear with quotes. The following statement

```xml
<Movie title = "Star Wars" year = "1977" genre = "sciFi" />
```

is a possible movie element in a document that conforms to this DTD.
11.3.4 Identifiers and References

Recall that certain attributes can be used as identifiers for elements.

1. In a DTD, we give these attributes the type **ID**.

2. Other attributes have values that are references to these *element ID’s*; these attributes may be declared to have type **IDREF**.

   The value of an **IDREF** attribute *must* also be the value of some **ID** attribute of some element, so the **IDREF** is in effect a *pointer* to the **ID**.

3. An alternative is to give an attribute the type **IDREFS**. In this case, the value of the attribute is a string consisting of a *list of ID’s*, separated by whitespace.

   The effect is that an **IDREF** attribute links its element to a *set of elements* – the elements identified by the **ID’s** on the list.
Example: The following figure shows a DTD in which stars and movies are given equal status, that is, both are subelements of the root element.

The **ID-IDREFS** correspondence is used to describe the many-many relationship between movies and stars.
11.3.4 Identifiers and References: Example

<!DOCTYPE StarMovieData [
  <!ELEMENT StarMovieData (Star*, Movie*)>
  <!ELEMENT Star (Name, Address+)>  
    <!ATTLIST Star
       starID ID #REQUIRED
       starredIn IDREFS #IMPLIED
    >
  <!ELEMENT Name (#PCDATA)>  
  <!ELEMENT Address (Street, City)>  
  <!ELEMENT Street (#PCDATA)>  
  <!ELEMENT City (#PCDATA)>  
  <!ELEMENT Movie (Title, Year)>  
    <!ATTLIST Movie
       movieID ID #REQUIRED
       starOf IDREFS #IMPLIED
    >
  <!ELEMENT Title (#PCDATA)>  
  <!ELEMENT Year (#PCDATA)> ]>
11.3.4 Identifiers and References: Example XML Document

```xml
<?xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Movie movieID = "sw" starsOf = "cf mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>
```
XML Schema
11.4 XML Schema

**XML Schema** is an alternative way to provide a schema for XML documents.

It is more powerful than DTD’s, giving the schema designer extra capabilities.

For example:

- XML Schema allows arbitrary restrictions on the number of occurrences of subelements.

- It allows us to declare types, such as integer or float, for simple elements.

- It gives us the ability to declare keys and foreign keys.
An **XML Schema** description of a schema is itself an XML document.

It uses the namespace at the URL:

```xml
http://www.w3.org/2001/XMLSchema
```

that is provided by the World-Wide-Web Consortium.

Each XML-Schema document thus has the form:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
</xs:schema>
```

In the following, we shall learn the most important tags from the XML-Schema namespace and what they mean.
An important component of schemas is the **element**, which is similar to an element definition in a DTD.

In the discussion that follows, you should be alert to the fact that, because XML-Schema definitions are XML documents, these schemas are themselves composed of “elements”. However, the elements of the schema itself, each of which has a tag that begins with **xs:**, are not the elements being defined by the schema.
The form of an element definition in XML Schema is:

```xml
<x$s:element name = "element name" type = "element type">
  constraints and/or structure information
</xs:element>
```

The element name is the chosen tag for these elements in the schema being defined.

The type can be either a simple type or a complex type.

- Simple types include the common primitive types, such as `xs:integer`, `xs:string`, and `xs:boolean`.
- There can be no subelements for an element of a simple type.
Example: Here are title and year elements defined in XML Schema:

```xml
<xsd:element name="Title" type="xsd:string" />
<xsd:element name="Year" type="xsd:integer" />
```

Each of these `<xs:element/>` elements is itself empty. But “Title” and “Year” in the defined document are not empty.
11.4.3 Complex Types

A complex type in XML Schema can have several forms, but the most common is a sequence of elements.

These elements are required to occur in the sequence given, but the number of repetitions of each element can be controlled by attributes \texttt{minOccurs} and \texttt{maxOccurs} that appear in the element definitions themselves.

The meaning of these attributes are as expected: \textit{No fewer than minOccurs occurrences of each element may appear in the sequence, and no more than maxOccurs occurrences of each element may appear.}

If there is more than one occurrence, they must all appear \textit{consecutively}.

The default, is one or both of these attributes are missing, is one occurrence.

To say that there is no upper limit on occurrences, use the value “unbounded” for \texttt{maxOccurs}.
11.4.3 Complex Types

The form of a definition for a complex-type that is a sequence of elements is shown below:

```xml
<xsd:complexType name = type name>
   <xsd:sequence>
      list of element definitions
   </xsd:sequence>
</xsd:complexType>
```

The name for the complex type is optional, but is needed if we are going to use this complex type as the type of one or more elements of the schema being defined.

An alternative is to place the complex-type definition between an opening `<xsd:element>` tag and its matched closing tag, to make that complex type be the type of the element.
Let us write a complete XML-Schema document that defines a very simple schema for movies:

- The root element for movie document will be `<Movies>`.
- The root will have zero or more `<Movie>` subelements.
- Each `<Movie>` element will have two subelements:
  - a `title`
  - a `year`

  in that order.

The XML-Schema document is shown below.
11.4.3 Complex Types: Example

1) `<? xml version = "1.0" encoding = "utf-8" ?>`
2) `<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">`

3) `<xs:complexType name = "movieType">
   4)   `<xs:sequence>
   5)     `<xs:element name = "Title" type = "xs:string"/>
   6)     `<xs:element name = "Year" type = "xs:integer"/>
   7)   </xs:sequence>
   8) </xs:complexType>

8) `<xs:element name = "Movies">
9)   `<xs:complexType>
10)   `<xs:sequence>
11)     `<xs:element name = "Movie" type = "movieType"
     minOccurs = 0 maxOccurs = "unbounded"/>
12)   </xs:sequence>
13) </xs:complexType>
14) </xs:element>

15) </xs:schema>
The previous XML Schema defines the same schema as the following DTD.

```
<!DOCTYPE Movies [ 
  <!ELEMENT Movies (Movie*)>
  <!ELEMENT Movie (Title, Year)>
  <!ELEMENT Title (#PCDATA) >
  <!ELEMENT Year (#PCDATA) >
] >
```
11.4.3 Complex Types

There are several other ways we can construct a complex type:

- In place of `xs:sequence` we could use `xs:all`, which means that each of the elements between the opening `<xs:all>` tag and its matched closing tag must occur, in any order, exactly once each.

- Alternatively, we could replace `xs:sequence` by `xs:choice`. Then, exactly one of the elements found between the opening `<xs:choice>` tag and its matched closing tag will appear.
11.4.3 Complex Types

The elements inside a sequence or choice can have minOccurs and maxOccurs attributes to govern how many times they can appear.

In the case of a choice, only one of the elements can appear at all, but it can appear more than once if it has a value of maxOccur greater than 1.

The rules for xs:all are different. It is not permitted to have a maxOccurs value greater than 1, but minOccurs can be either 0 or 1. In the former case, the element might not appear at all.
A complex type can have attributes. That is, when we define a complex type $T$, we can include instances of element `<xs:attribute>`.

When we use $T$ as the type of an element $E$, $E$ can have (or must have) an instance of this attribute.

The form of an attribute definition is:

$\langle xs:attribute \text{ name} = \text{attribute name} \text{ type} = \text{type name} \\
\text{other information about the attribute/} \rangle$

The “other information” may include information such as

- a default value,
- usage (*required* or *optional* – the latter is the default).
Example: The notation

<xsl:attribute name="year" type="xs:integer" default="0"/>

defines year to be an attribute of type integer. The default value of year is 0.
Example: The notation

<xss:attribute name="year" type="xs:integer" use="required" />

defines year to be an attribute of type integer. Any element of the type being defined must have a value for attribute year.
11.4.4 Attributes

Attribute definitions are placed within a complex-type definition.

In the following example, we make the type `movieType` have attributes for `title` and `year`, instead of as subelements.
11.4.4 Attributes: Example

1)  <? xml version = "1.0" encoding = "utf-8" ? >
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema" >

3)  <xs:complexType name = "movieType" >
4)      <xs:attribute name = "Title" type = "xs:string"
             use = "required" / >
5)      <xs:attribute name = "Year" type = "xs:integer"
             use = "required" / >
6)  </xs:complexType>

7)  <xs:element name = "Movies" >
8)      <xs:complexType>
9)          <xs:sequence>
10)     <xs:element name = "Movie" type = "movieType"
                minOccurs = 0 maxOccurs = "unbounded" / >
11")     </xs:sequence>
12)  </xs:complexType>
13)  </xs:element>

14)  </xs:schema>
11.4.4 Attributes: Example

The previous XML Schema defines the same schema as the following DTD.

```
<!DOCTYPE Movies [  
  <!ELEMENT Movies (Movie*)>  
  <!ELEMENT Movie EMPTY>  
    <!ATTLIST Movie  
      title CDATA #REQUIRED  
      year CDATA #REQUIRED  
    >  
  ]>  
```
11.4.5 Restricted Simple Types

It is possible to create a restricted version of a simple type such as integer or string by limiting the values the type can take.

These types can then be used as the type of an attribute or element.

We shall consider two kinds of restrictions here:

1. Restricting numerical values by using `minInclusive` to state the lower bound, and `maxInclusive` to state the upper bound.

2. Restricting values to an enumerated type.
11.4.5 Restricted Simple Types

The form of a range restriction is shown below. The restriction has a base, which may be a primitive type (e.g., `xs:string`) or another simple type.

```xml
<xs:simpleType name = type name>
  <xs:restriction base = base type>
    upper and/or lower bounds
  </xs:restriction>
</xs:simpleType>
```
Suppose we want to restrict the year of a movie to be no earlier than 1915. We can define a simple type as follows. The type `movieYearType` can then be used to replace `xs:integer` in our previous XML Schema example.

```xml
<x:simpleType name = “movieYearType”>
  <xs:restriction base = “xs:integer”>
    <xs:minInclusive value = “1915”/>
  </xs:restriction>
</xs:simpleType>
```
Our second way to restrict a simple type is to provide an enumeration of values.

The form of a single enumerated value is:

\[
\langle xs:enumeration value = \textit{some value} / \rangle
\]

A restriction can consist of any number of these values.
Example: Let us design a simple type suitable for the genre of movies. In our running example, we have supposed that there are only four possible genres: comedy, drama, sciFi, and teen. The following example shows how to define a type genreType that could serve as the type for an element or attribute representing our genres of movies.

```xml
<xs:simpleType name = "genreType">
   <xs:restriction base = "xs:string">
      <xs:enumeration value = "comedy"/>
      <xs:enumeration value = "drama"/>
      <xs:enumeration value = "sciFi"/>
      <xs:enumeration value = "teen"/>
   </xs:restriction>
</xs:simpleType>
```
An *element* can have a **key** declaration, which says that when we look at a certain class $C$ of elements, values of one or more given *fields* within those elements are unique.

The concept of **field** is actually quite general, but the most common case is for a field to be either

1. A subelement, or

2. An attribute.

The class $C$ of elements is defined by a **selector**.

Like fields, selectors can be complex, but the most common case is a **sequence of one or more element names, each a subelement of the one before it**. In terms of a tree of semistructured data, the class is all those nodes reachable from a given node by following a particular sequence of arc labels.
Example: Consider a tree structure where “star” nodes are children of the root node. Suppose we want to say that among all the nodes we can reach from the root by following a *star* label, which we find following a further *name* label leads us to a unique value. (That is, for all those children of the root node that are “star” nodes, their *names* should all be different.)

Then, for the key declaration, the **selector** would be *star* and the **field** would be *name*. The implication of asserting this key is that within the root element shown, there cannot be two stars with the same name.
Example (continued):
Now assume that the root node can also have another type of children called “movie”. If movies had names instead of titles, then the key assertion would NOT prevent a movie and a star from having the same name (because their selectors are different). Moreover, if there exist two “star” nodes in the tree but they have different selectors (that is, they are at different positions of the tree), those two stars can still have the same name (without violating the key constraint).
11.4.6 Keys in XML Schema

The form of a **key declaration** is:

```xml
<xs:key name = key name>
  <xs:selector xpath = path description/>
  <xs:field xpath = path description/>
</xs:key>
```

There can be more than one line with an **xs:field** element, in case several fields are needed to form the key.
An alternative is to use the element `xs:unique` in place of `xs:key`.

- The difference is that if `key` is used, then the field must exist for each element defined by the selector.
- However, if `unique` is used, then they might not exist, and the constraint is that they are unique if they exist.
11.4.6 Keys in XML Schema

The **selector path** can be any sequence of elements, each of which is a subelement of the previous. The element names are separated by **slashes**.

The field can be any **subelement** of the last element on the selector path, or it can be any **attribute** of that element.

- If it is an attribute, then it is preceded by the @ sign.

There are other options, and in fact, the selector and field can be any XPath expressions (for details, see Chapter 12.1).
11.4.6 Keys in XML Schema: Example

1)  
   <? xml version = "1.0" encoding = "utf-8" ?>

2)  
   <xs: schema xmlns: xs = "http : //www.w3.org/2001/XMLSchema" >

3)  
   <xs:simpleType name = "genreType">  

4)  
   <xs:restriction base = "xs:string">  

5)  
   <xs:enumeration value = "comedy" / >

6)  
   <xs:enumeration value = "drama" / >

7)  
   <xs:enumeration value = "sciFi" / >

8)  
   <xs:enumeration value = "teen" / >

9)  
   </xs:restriction>

10)  
    </xs:simpleType>

11)  
    <xs:complexType name = "movieType">  

12)  
    <xs:sequence>

13)  
    <xs:element name = "Title" type = "xs:string" / >

14)  
    <xs:element name = "Year" type = "xs:integer" / >

15)  
    <xs:element name = "Genre" type = "genreType"  
                minOccurs = "0" maxOccures = "1" / >

16)  
    </xs:sequence>

17)  
    </xs:complexType>
11.4.6 Keys in XML Schema: Example

18)  
   <xs:element name = “Movies”>

19)  
    <xs:complexType>

20)  
    <xs:sequence>

21)  
    <xs:element name = “Movie” type = “movieType”
                 minOccurs = “0” maxOccurs = “unbounded” />

22)  
    </xs:sequence>

23)  
    </xs:complexType>

24)  
    <xs:key name = “movieKey”>

25)  
    <xs:selector xpath = “Movie” />

26)  
    <xs:field xpath = “Title” />

27)  
    <xs:field xpath = “Year” />

28)  
    </xs:key>

29)  
    </xs:element>

30)  
    </xs:schema>
Note that in line 24, the key is given a name `movieKey`. This name will be used if it is referenced by a `foreign key`; otherwise, the name is irrelevant.

The selector path is just `Movie`, and there are two fields: `Title` and `Year`.

The meaning of this key declaration is that, within any `Movies` element, among all of its `Movie` subelements, no two can have both the same title and the same year, nor can any of these values be missing.
11.4.7 Foreign Keys in XML Schema

We can also declare that an element has, perhaps deeply nested within it, a field or fields that serve as a reference to the key for some other element. This is the foreign-key constraint.

The form of a **foreign key** definition in XML Schema is:

```
<xs:keyref name = foreign-key name refer = key name>
  <xs:selector xpath = path description/ >
  <xs:field xpath = path description/ >
</xs:keyref>
```

The schema element is **xs:keyref**. The foreign-key itself has a name, and it refers to the name of some key or unique value.

The selector and field(s) are as for keys.
1)    <!-- xml version = "1.0" encoding = "utf-8" -->
2)    <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema" >
3)    <xs:element name = "Stars" >
4)        <xs:complexType>
5)            <xs:sequence>
6)                <xs:element name = "Star" minOccurs = "1"
                        maxOccurs = "unbounded" >
                <xs:complexType>
8)                    <xs:sequence>
9)                        <xs:element name = "Name"
                                type = "xs:string" / >
10)                       <xs:element name = "Address"
                                type = "xs:string" / >
11)                       <xs:element name = "StarredIn"
                                minOccurs = "0"
                                maxOccurs = "unbounded"
11.4.7 Foreign Keys in XML Schema: Example

12) `<xs:complexType>`
13) `  <xs:attribute name = “title”
    type = “xs:string”/>`
14) `  <xs:attribute name = “year”
    type = “xs:integer”/>`
15) `</xs:complexType>`
16) `</xs:element>`
17) `</xs:sequence>`
18) `</xs:complexType>`
19) `</xs:element>`
20) `</xs:sequence>`
21) `</xs:complexType>`
22) `  <xs:keyref name = “movieRef” refers = “movieKey”>`
23) `    <xs:selector xpath = “Star/StarredIn”/>`
24) `  <xs:field xpath = “@title”/>`
25) `  <xs:field xpath = “@year”/>`
26) `</xs:keyref>`
27) `</xs:element>`
11.4.7 Foreign Keys in XML Schema: Example

Line (22) through (26) define a foreign key.

In line (22) we see that the name of this foreign-key constraint is `movieRef` and that it refers to the key `movieRef` that was defined in the previous example.

Note that this foreign key is defined within the `<Stars>` definition. The selector is `Star/StarredIn`. That is, it says we should look at every `<StarredIn>` subelement of every `<Star>` subelement of a `<Stars>` element.

From that `<StarredIn>` element, we extract the two fields `title` and `year`. The @ indicates that these are attributes rather than subelements.

The assertion made by this foreign-key constraint is that any title-year pair we find in this way will appear in some `<Movie>` element as the pair of values for its subelements `<Title>` and `<Year>`. 