XML Schema to ANTLR Grammar mapping

The image files and examples in this document were made with the help of a Company.xsd file from 101companies. Source: https://github.com/101companies/101repo/blob/master/contributions/xsdDataSet/xsd/Company.xsd (Last access: March 30th 2016 at 14:37 UTC+2)

The task was to implement an automated mapping from XML Schema Definition to ANTLR grammar for a Domain Specific Language similar to JSON. Additionally, an automated mapping from XML (instances of certain XSDs) to instances in the new DSL was implemented.

The tools and languages used were:

Language	**Role**
Python (version 3.4.3)	for the mapping scripts
ANTLR (version 4.5.1)	target language of the mapping
XML Schema Definition (a subset thereof)	source language of the mapping
XML	XSD files are written in XML; instances of XSD files are written in XML
Java	ANTLR grammar files are compiled to Java files for the creation of parser and lexer objects
“myDSL“	JSON-like domain specific language which describes instances of XSDs
Technology	**Role**
(ANTLR) lexer, parser	checks correctness of a “myDSL“ file in regard to an ANTLR grammar generated from a XSD
ElementTree	Python package for parsing and representing XML (and XSD, respectively) in Python

The XSD subset used consists of the following features:

Feature	Supported by the mapper	Limitations
attribute, complexType, element, sequence	**yes**	
namespaces	no	
“default” attribute	no	
“fixed” attribute	**yes**	

restrictions	no	
extensions	no	
simpleContent	no	
all, choice	no	
maxOccurs, minOccurs	**yes**	multiplicity indicators for ambiguous combinations are chosen rather arbitrarily
element groups	no	
any, anyAttribute	no	
substitutionGroup	no	

Supported built-in XSD datatypes:

Type	**Limitations**
xs:boolean	
xs:decimal	no range checking
xs:double	no range checking; is treated as float
xs:float	no range checking
xs:int (derived datatype)	no range checking
xs:string	only alphabetical letters; if a string has whitespace in it, it is considered as multiple strings

“no range checking” means that the ANTLR parser does not check whether a given number is in the range of the alleged data type. E.g., a double must be smaller than 2^53 (lexical representation: 2e53), however, the generated ANTLR grammar will accept greater values for a double. Range checking should be done in the compilated Java parser.

The code

The code is divided into **four Python files**.

File name	Purpose
xsd2ebnf_parser.py	API and string manipulation methods for parsing XSD files or nodes to ANTLR grammar files
xsd2ebnf.py	contains: start/end line of ANTLR grammars; dictionary for mapping XSD types to ANTLR rules; dictionary for easy checking of XML element types
xml2dsl_parser.py	mapping from XSD instances in XML format to myDSL format
helper.py	different simple methods, e.g. to map multiplicity indicators in XSD to their ANTLR form, checking whether a node is a “simple” node, etc.

Parsing an XSD file

In the course of parsing an XSD file to an ANTLR grammar file, the nodes of the XSD are mapped to strings which are appended to the list object `antlr4_grammar`. Type rules are appended to the set object `type_rules`.

The parsing of an XSD file is started with the **`parse_xsd(path_to_file)`** method, where `path_to_file` is a valid XSD file.
`parse_xsd(path_to_file)` calls the `make_grammar(path_to_file)` method. All methods starting with `parse` are part of the parsing API, while the methods starting with `make` perform string operations.

`make_grammar(path_to_file)` parses the file to an `xml.etree.ElementTree.ElementTree` object. After appending the grammar name, based on the name of the parsed file, to `antlr4_grammar`, all nodes of the root element of the XSD file are iterated and parsed with the function `parse_node(node)`, where node is an `xml.etree.ElementTree.Element` object. The root element of a valid XSD file is an `<xs:schema>` node, which does not need to be parsed.
The ANTLR mappings of the parsed nodes are appended to `antlr4_grammar` or `type_rules`, and after they have all been processed, `make_grammar()` appends `type_rules` as well as a rule for skipping whitespace, tabs etc. to `antlr4_grammar`. The resulting list's elements are then concatenated and returned.

`parse_node(node)` either passes to the node to `parse_simple_node(node)` – in the case the node has no childred, but a `type` attribute or is a reference (`ref`) – or to `parse_complex_node(node)` in any other case. The exception are `complexType` elements without a `name` attribute and `sequences`; in that case, they are skipped, and their children are parsed. (Please note that the order of elements is by default fixed in the generated ANTLR grammar, and thus, corresponds to the purpose of `sequence` in XML.) In the case of parsing complex nodes or the children elements of `sequence`- or `complexType` elements, we keep track of the number of open elements in a global variable `open_ele_cnt`. See the paragraph about `parse_complex_node(node)` for more information.

Example for `parse_node(node)`

```
<xs:element name="Department"> (1)
  <xs:complexType> (2)
    <xs:sequence> (3)
      <xs:element ref="Name"/> (4)
      <xs:element name="Manager" type="Employee"/> (5)
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="Employee"> (6)
  <xs:sequence>
    ...
  </xs:sequence>
</xs:complexType>
```

(1) parsed by `parse_complex_node`, since it has children
(2), (3) skipped; children will be parsed
(4) parsed by `parse_simple_node`, since it has no children and a `ref` attribute
(5) parsed by `parse_simple_node`, since it has no children and a `type` attribute
(6) parsed by `parse_complex_node`, since it is complexType, but has a `name` attribute

`parse_simple_node(node)` reads the attributes of the `node` and passes them to `make_simple_element(...)` for the ANTLR String generation. If the element has a type attribute, it passes that type to `make_type_rule(...)` to handle it. See the paragraph about `make_type_rule(...)` for more information.

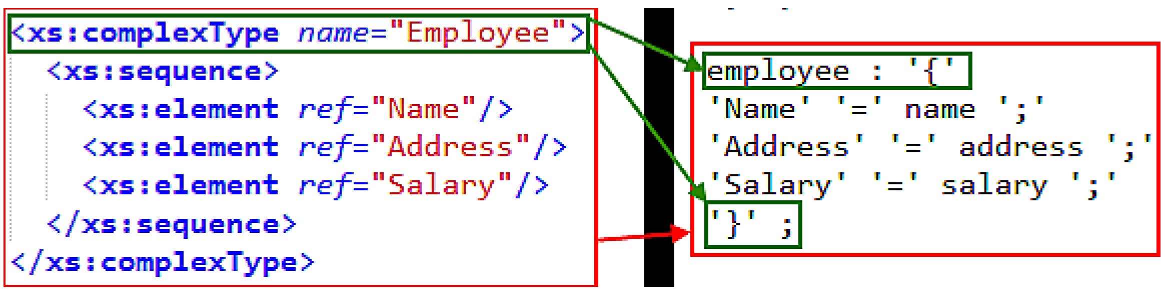
`make_simple_element(name, type_, indicator=None, multiplicity=None)` generates and appends a string of the ANTLR grammar to antlr4_grammar. There are four different kinds of strings it can generate, depending on whether an indicator and a multiplicity are passed as arguments. It is also important to notice that the operator and the semicolon vary slightly depending on whether this element is a child of another element or the direct child of the root <xs:schema>. See the example below for clarification.

Example for `parse_simple_node(node)` + `make_simple_element(...)` + `make_type_rule(...)`

```
node = <xs:element name="Name" type="xs:string" />
→ parse_simple_node(node)
  → make_type_rule("xs:string")
      → "STRING : ([a-zA-Z])+ ;"
  → make_simple_element(age, INT, None, None)
      → "name : STRING+ ;"

(the next node is a child of an element other than the root
element)
node = <xs:element name="Manager" type="Employee"/>
→ parse_simple_node(node)
  → make_type_rule("Employee")
    → (do not generate type rule, but return value for
make_simple_element:) "employee"
  → make_simple_element(Manager, employee, None, None)
    → "'Manager' '=' employee ';'"


(the next node is a child of an element other than the root
element)
node = <xs:element maxOccurs="unbounded" minOccurs="0"
ref="Department"/>
→ parse_simple_node(node)
  → make_simple_element(Department, department, None, *)
    → ('Department' '=' department ';')*
```

Please have a look at the page after the next for a more visual, non-verbose representation of this mapping.

`parse_complex_node(node)` creates the "head" (name of node and opening curly brace) of the node via `make_complex_element_head(name, multiplicity)`, parses all children of the node, and creates a "foot" (closing curly brace and semicolon) of the node via `make_complex_element_foot(multiplicity)`. It also checks whether the node has multiplicity defined and adjusts head and foot respectively. Like `parse_node(node)`, it also keeps track of how many elements are currently opened.

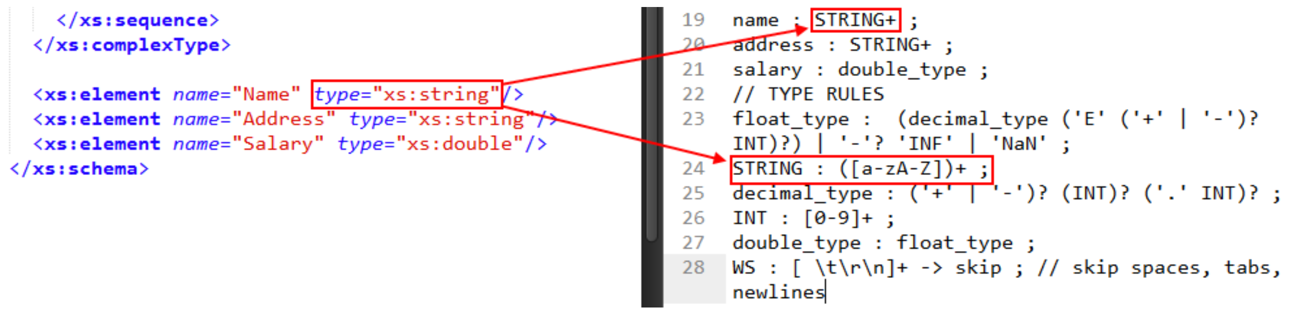


```
<xs:complexType name="Employee">
  <xs:sequence>
    <xs:element ref="Name"/>
    <xs:element ref="Address"/>
    <xs:element ref="Salary"/>
  </xs:sequence>
</xs:complexType>
```

```
employee : '{'
'Name' '=' name ';'
'Address' '=' address ';'
'Salary' '=' salary ';'
'}' ;
```

Picture 1: Example of make_complex_rule(...)

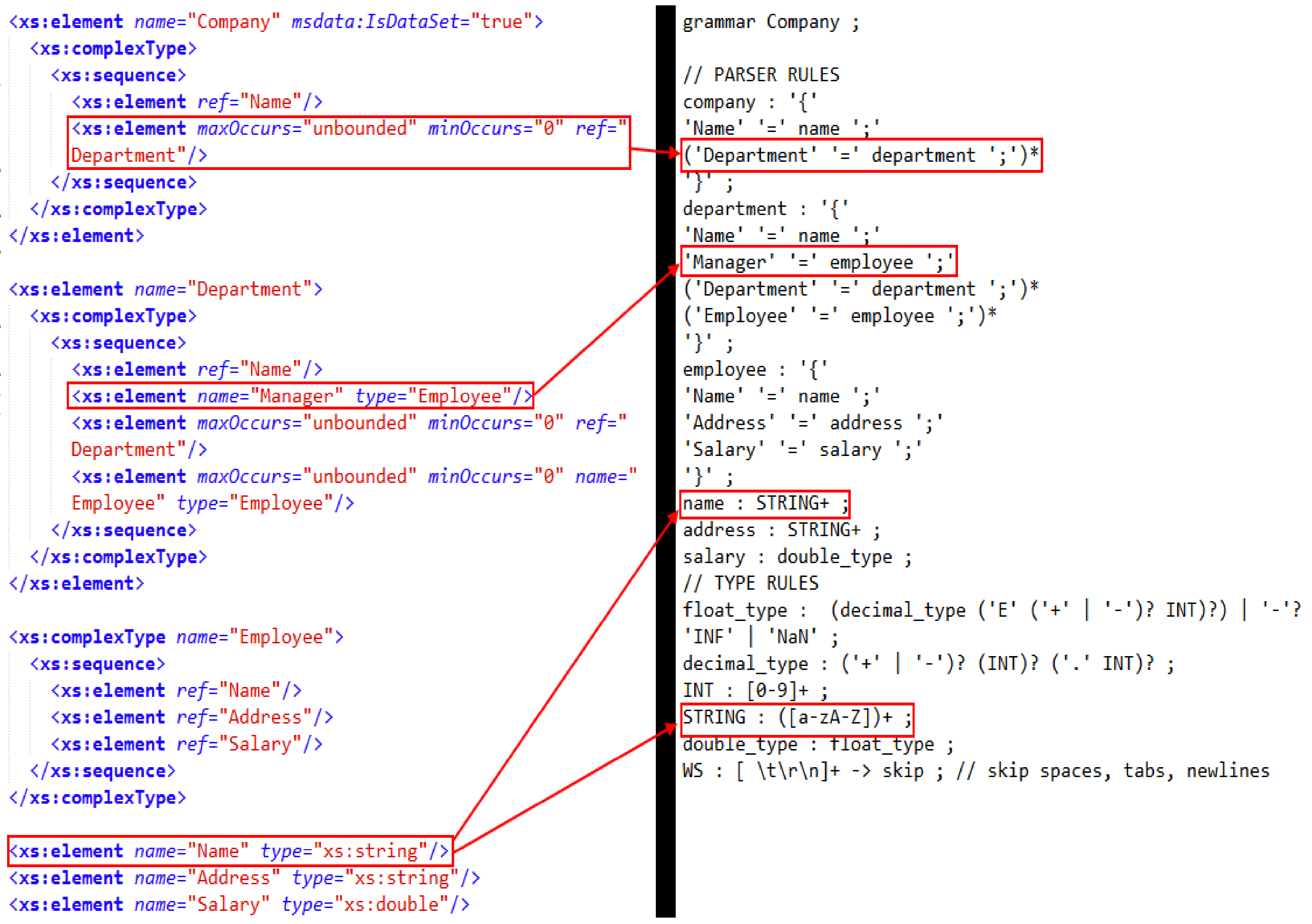
xsd2ebnf (the dictionary in the Python file of the same name) is used to map a XSD datatype to 1) its representation in the body of a parser rule 2) its type rule 3) any dependencies to other type rules. (For example, the ANTLR representation for `xs:decimal` relies on the definition on `xs:int`, so if a decimal type from an XSD is mapped, a type rule must be generated for int as well.)



```
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Name" type="xs:string"/>
  <xs:element name="Address" type="xs:string"/>
  <xs:element name="Salary" type="xs:double"/>
</xs:schema>
```

```
name : STRING+ ;
address : STRING+ ;
salary : double_type ;
// TYPE RULES
float_type :  (decimal_type ('E' ('+' | '-')?
INT)?) | '-'? 'INF' | 'NaN' ;
STRING : ([a-zA-Z])+ ;
decimal_type : ('+' | '-')? (INT)? ('.' INT)? ;
INT : [0-9]+ ;
double_type : float_type ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs,
newlines
```

Picture 2: Example of type mapping



```
<xs:element name="Company" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Name"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="
      Department"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Department">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Name"/>
      <xs:element name="Manager" type="Employee"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="
      Department"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="
      Employee" type="Employee"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="Employee">
  <xs:sequence>
    <xs:element ref="Name"/>
    <xs:element ref="Address"/>
    <xs:element ref="Salary"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Name" type="xs:string"/>
<xs:element name="Address" type="xs:string"/>
<xs:element name="Salary" type="xs:double"/>
```

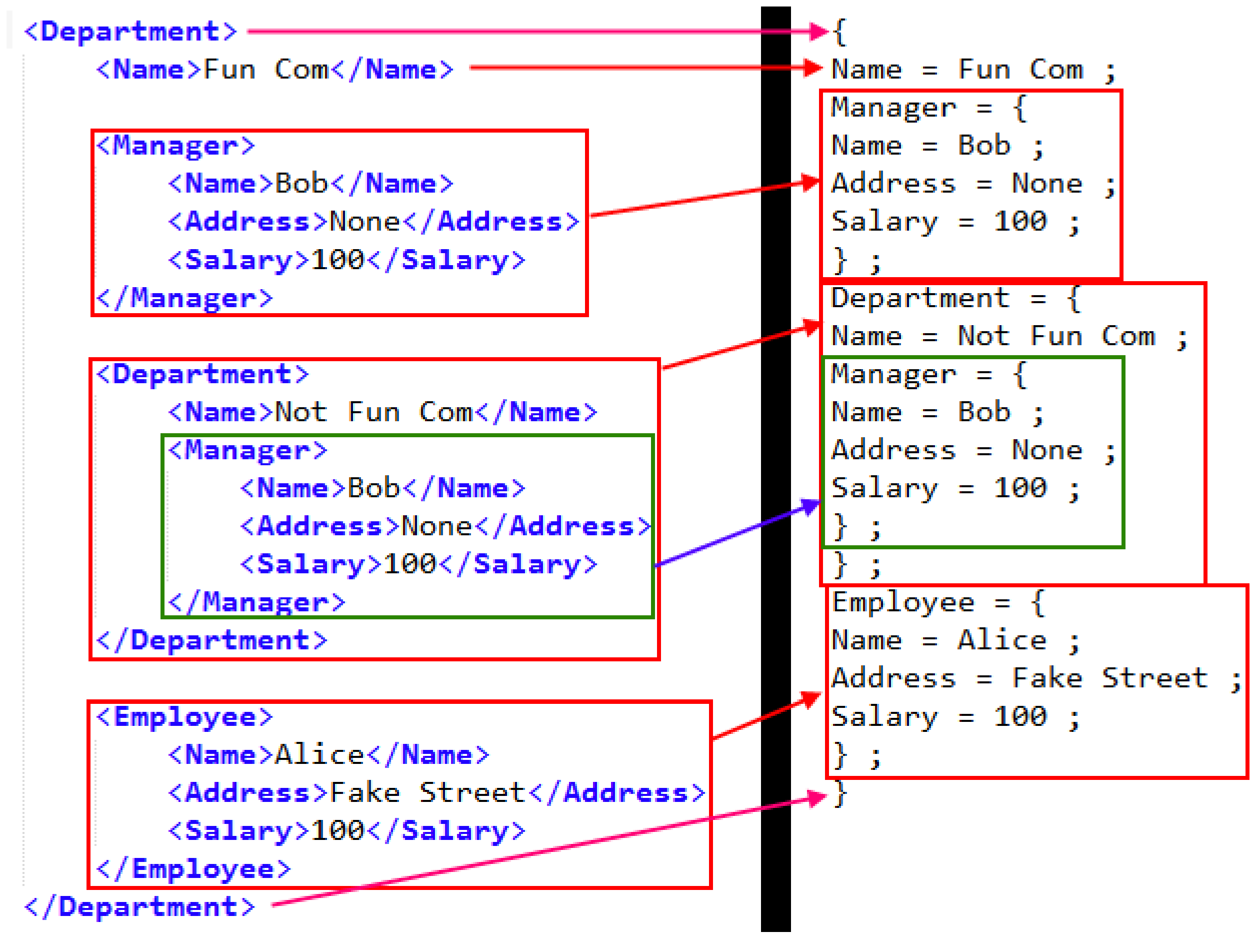
```
grammar Company ;

// PARSER RULES
company : '{'
'Name' '=' name ';'
('Department' '=' department ';')*
'}' ;
department : '{'
'Name' '=' name ';'
'Manager' '=' employee ';'
('Department' '=' department ';')*
('Employee' '=' employee ';')*
'}' ;
employee : '{'
'Name' '=' name ';'
'Address' '=' address ';'
'Salary' '=' salary ';'
'}' ;
name : STRING+ ;
address : STRING+ ;
salary : double_type ;
// TYPE RULES
float_type :  (decimal_type ('E' ('+' | '-')? INT)?) | '-'?
'INF' | 'NaN' ;
decimal_type : ('+' | '-')? (INT)? ('.' INT)? ;
INT : [0-9]+ ;
STRING : ([a-zA-Z])+ ;
double_type : float_type ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Picture 3: Examples of make_simple_rule(...)

XML instance parser

The parser from XML instances to myDSL instances can be found in `xml2dsl_parser.py`. It looks very similar to the XSD to ANTLR parser, albeit a lot simpler. Simple nodes – i.e. nodes without children and without attributes – are simply parsed as
“`node name = node text ;`”
while complex nodes are parsed as
“`node name {…}`”

Special cases that have to be handled are among others that top-level elements (like the `Department` element which wraps all other elements of the XML instance in the example below) are nameless in myDSL notation.



```
<Department>
    <Name>Fun Com</Name>

    <Manager>
        <Name>Bob</Name>
        <Address>None</Address>
        <Salary>100</Salary>
    </Manager>

    <Department>
        <Name>Not Fun Com</Name>
        <Manager>
            <Name>Bob</Name>
            <Address>None</Address>
            <Salary>100</Salary>
        </Manager>
    </Department>

    <Employee>
        <Name>Alice</Name>
        <Address>Fake Street</Address>
        <Salary>100</Salary>
    </Employee>
</Department>
```

```
{
Name = Fun Com ;
Manager = {
Name = Bob ;
Address = None ;
Salary = 100 ;
} ;
Department = {
Name = Not Fun Com ;
Manager = {
Name = Bob ;
Address = None ;
Salary = 100 ;
} ;
} ;
Employee = {
Name = Alice ;
Address = Fake Street ;
Salary = 100 ;
} ;
}
```

Some implementation decisions worth mentioning

The problem with Strings

If I were to use quotation marks around Strings, the ANTLR grammar would be much clearer:

```
"STRING: '\"' (~'\"')* '\"' ; "
```

However, I would run into problems when converting XML-instances to DSL-instances.
In XML instances, I have no way to look up the type of an object directly from the XML file.

```
<Employee>
	<Name>Bob Bobson</Name>
	<Address>123 Fake Street</Address>
	<Salary>3000</Salary>
</Employee>
```

In this example, the salary object could be either an int, a String, or some other data type. If I would use quotation marks around Strings in my DSL, I would have to determine the object's type and decide whether it is a String, and thus needs quotation marks, or not. To solve this problem, I just dropped quotation marks altogether, which makes the String generation for my DSL very limited:

```
"STRING : ([a-zA-Z])+ ;"
```

If I were to add numbers to it, like

```
"STRING : ([a-zA-Z0-9])+ ;"
```

I would run into a new problem: I would have to keep track of the order of the lexer rules, so that the rule for int is always mentioned before the String rule. Otherwise pure numerical strings would always be interpreted as Strings instead of int. I dropped this feature for the sake of simplicity of the code, even though to would be easy to implement it in the form of a “hack” for this special case: before adding the lexer rules to the grammar String object, check each String object in the set of lexer rules, check if it matches the String for ANTLR String objects, and postpone its addition to the grammar String until the int String has been added.

minOccurs/maxOccurs

Exact values, like e.g. minOccurs = 0 and maxOccurs = 10 are not supported; however, any combination of minOccurs ϵ [0, 1] and maxOccurs ϵ [1, unbounded] is supported. Elements which have only one of those attributes are also supported to a certain extent. maxOccurs = 1 is interpreted as a choice (“?” in ANTLR), maxOccurs = “unbounded” is interpreted as at least one element (“+”). minOccurs = 0 and minOccurs = 1 are interpreted as “*” and “+”, respectively. These interpretations should however be adapted to the context of the used XSD, since they are ambiguous.
All other combinations and values of minOccurs and/or maxOccurs are ignored, i.e. the element will be treated as a single required element by ANTLR.