AMS 601.3 language

Table of Contents

I. Introduction

We will be using JSON to encode the data we want to represent. It is perfectly defined here with state machines: http://www.json.org/

If using Python 3, you can found a built-in module parsing Json :
http://docs.python.org/3/library/json.html

Here is for instance the definitions of a value, a json object (we will always use this term to distinguish from an object in our language), and an array.

value
- *string*
- *number*
- *object*
- *array*
- **true**
- **false**
- **null**

object
- {}
- { *members* }

members
- *pair*
- *pair* , *members*

pair
- *string* : *value*

array
- []
- [*elements*]

elements
- *value*
- *value* , *elements*

II. First definitions

We are going to use the same method to describe our grammar. For instance, mathematically, we understand the definition of *value* as follows : a word m is verifying the state machine *value* if and only if m matches one of the following other state machine (*string*, *number*, *object*, *array*, **true**, **false**, **null**).

We refer to a state machine as a pattern. For instance, *value* is a pattern. We will use the term 'm matches the pattern A' or simply 'm matches A' if the word m is verifying the state machine A. For instance “car” matches *value.*

We define *NonEmptyString* as the same pattern as *string*, except that it does not match the empty string (“”).

A word matches *list(A)* if it is a comma separated list of words that matches the pattern A. The formal definition is:

list(A)

empty word (i.e. nothing)

elements_A

elements_A

A

A, elements_A

We also define *NonEmptyList(A)* as:

NonEmptyList(A)

elements_A

III. Syntax of the Rauzy language

Note that all characters used in the syntax are case sensitive

1. Formal definition

Both relations and Rauzy objects (we will use this term to refer to the objects we are describing in our language) are represented by Json objects. To distinguish them, a member “nature” will have a specific value (“relation” and “object” respectively).

For A a pattern, we define:

named(A)

NonEmptyString : A

Here is the grammar of a Rauzy object (the definition of a relation follows):

{

“nature” : “object”,

“extends” : string,

“objects” : { *list(Named(Rauzy_object))* },

“relations” : { *list(Named(relation))* },

“properties” : { *list(Named(string))* },

“library” : string

}

The members “extends”, “objects”, “relations”, “properties” and “library” can be empty (e.g. “objects : {}), **null** (e.g. “relations” : **null**), or even not defined. In theses cases, it will be considered as empty.

Here is the grammar of a relation :

{

“nature” : “relation”,

“extends” : string,

“from” : [*list(string)*],

“to” : [*list(string)*],

“directional” : **true** or **false**,

“properties” : { *list(Named(string))* }

}

The members “extends” and “properties” can be empty (e.g. “objects : {}), **null** (e.g. “extends” : **null**), or even not defined. In theses cases , it will be considered as empty.

A library file will be as follows:

{

“nature” : “library”,

“relations” : { *list(Named(relation))* },

“objects” : { *list(Named(Rauzy_object))* }

}

2. Backus-Naur Form (BNF)

This form may have less precision than the other since it does not include all the conventions in json (you may have multiple spaces, have multiple definition of the same term etc). The formal definition is the reference.

<*NonEmptyString*> := ' “ ' (<*Alpha*> | <*Digit*>)+ ' ” '
<*String*> ::= ' “ '(<*Alpha*> | <*Digit*>)* ' ” '
<*Alpha*> ::= [a_zA-Z]
<*Digit*> ::= [0-9]

<*List*(<*A*>)> ::= (<*A*> ',' <*A*>)*
<*NonEmptyList*(<*A*>)> ::= <A> (',' <*A*>)*

<*Named*(<*A*>)>::= <*NonEmptyString*> ':' <*A*>

We define recursively:
<*Unordered*(X)> ::=X
<*Unordered*(X, List)> ::= (X ',' <*Unordered(*List)>) | (<*Unordered(*List)> ',' X)

<*RauzyObject*> ::= '{' <*Unordered*(
' “nature” ' ':' ' “object” ' ,
(' “extends” ' ':' <S*tring*>)?,
(' “objects” ' ':' '{' <L*ist(<Named(RauzyObject*)>)> '}')?,
(' “relations” ' ':' '{' <L*ist(<Named(Relation)>)>* '}')? ,
(*'* “properties” ' ':' '{' <L*ist(<Named(String)>)>* '}')? ,

(' “library” ' ':' <string>)?) '}'

<*RauzyRelation*> ::= '{' <*Unordered*(
' “nature” ' ':' '“relation” ' ,
(' “extends” ' ':' <S*tring*>)? ,
' “from” ' ':' '[' <*List*(*String*)> ']' ,
' “to” ' ':' '[' <*List*(*String*)> ']' ,
'“directional” ' ':' 'True' | 'False' ,
(' “properties” ' ':' '{' <L*ist(<Named(String)>)*> '}')?)
'}'

<*Library*> := '{' <*Unordered*(
' “nature” ' ':' '“library” ' ,
(' “relations” ' ':' <S*tring*>)? ,
(' “objects” ' ':' '[' <*List*(<*Named*(<*RauzyRelation*>)>)> ']')? ,
(' “objects” ' ':' '[' <*List*(<*Named*(<*RauzyObject*>)>)> ']')?)
'}'

IV. Semantics of the Rauzy language

We reference here RFC 2119.

1. MUST: This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

2. MUST NOT: This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

3. SHOULD: This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. SHOULD NOT: This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

0. Parsing rules

The absence of mandatory keys in the json objects or arrays MUST raise an error. The presence of other keys not defined in this standard MUST NOT raise an error. In particular it can be used to develop extensions.

Note that all characters used in the syntax are case sensitive

1. Loading a file

When loading a file (i.e. interpreting the content of a file as an object described using the Rauzy

language), the “library” member of the root object, if defined, MUST be used to import the prototypes/classes defined in the library file. The “library” value MUST be used as a relative path from the location of the file.

1.1 Loading the library

When loading a library, two environments are created, one for the object classes and one for the relation classes. Every pair (*name*, *obj*) in the list of named objects MUST create in the environment for objects classes the association between the string *name* and the object *obj*. The set of names will be referred to as the environment of classes or simply classes if it is not ambiguous.

Similarly, every pair (*name*, *relation*) in the list of named relations MUST create in the environment for relation classes the association between the string *name* and the relation *relation*. The set of names will be referred to as the environment of relations or simply classes if it is not ambiguous.

The objects defined in the list of named objects MUST be able to refer to relations extending relations present in the environment of relations. For that reason, the list of named relations SHOULD be parsed first.

1.1.1 Loading the relations

The relations present in the list of named relations, SHOULD have empty values for the “from” and “to” fields since their represent names of objects and that no objects are defined in this scope.

1.1.2 Loading of the objects

It MUST be possible for a class A to extend an other class B of the list, without any consideration for the order of A and B in the list of named objects. Also, it MUST be possible for a class A to include objects extending a class B. Thus, the loading of the object classes MUST take into account dependencies and work as long as there are no cyclic dependencies. In the case of cyclic dependencies, an error SHOULD be raised.

1.2 Loading an object

We suppose that the environment of object classes and relation classes is defined.

It is recommended that the presence of a non-empty “library” member in an object that is not the root object, does not raise an error.

The relations can refer to objects contained in the object one is currently parsing. Then, the relations SHOULD be parsed after the parsing of the “objects” field.

When parsing relations, they can refer to names of objects defined in the object that is currently parsed, or in descendants of these objects. Thus, the name SHOULD not be ambiguous (i.e. exist twice in the descendant or the contained objects). If the name is ambiguous, the behaviour is not defined.

When creating an object A that extends B:

- B MUST be present in the environment of object classes (i.e. must have been defined in the libray file). If not, this SHOULD raise an error like 'Reference to an undefined class B'.

- only properties can be added in A. One SHOULD raise an error if the fields “objects” or “relations” are not empty in the json description. Indeed, the only objects and relations are the one that come from the object class associated to the name B

V. Mandatory deliverables

The program parsing and interpreting the here-above languages must be programmed in Python. I (J-B) is suggesting Python 3.

The program must be able to :

- load a Json file into an abstract representation of it

- allow modifications on the abstract representation (add/remove/modify/add properties to/ Rauzy objects or relations, and the same manipulations on the library associated to a model – add/rename object or relation class).

- save an abstract object into a Json file with its library

- provide abstraction, comparison and flattening functions that are implementation dependent

Moreover, a report containing explanations – in particular of the implementation dependent parts-, a tutorial and some examples must be provided.