Les arbres comme modèle de données
JSON et JSON Path

E.Coquery

emmanuel.coquery@univ-lyon1.fr

http://emmanuel.coquery.pages.univ-lyon1.fr/
→ Enseignement → BD NoSQL

Données arborescentes

Données stockées dans un arbre :

- différent du modèle relationel
- données de base ($\approx$ feuilles) : textuelles (mais interprétables)
- nœuds “de structure” ($\approx$ internes) : différents types selon le modèle

Deux standards (modèles) majeurs :

- **JSON**
- XML

Modèle JSON

Nœuds :

- données de base (atomiques) : syntaxe
  - texte (`string`) "toto"
  - nombre (`number`) 2.5
  - `null` null
- tableau (`array`) [... , ... , ...]
- dictionnaire (`object`) { "a" : ... , "truc" : ... }

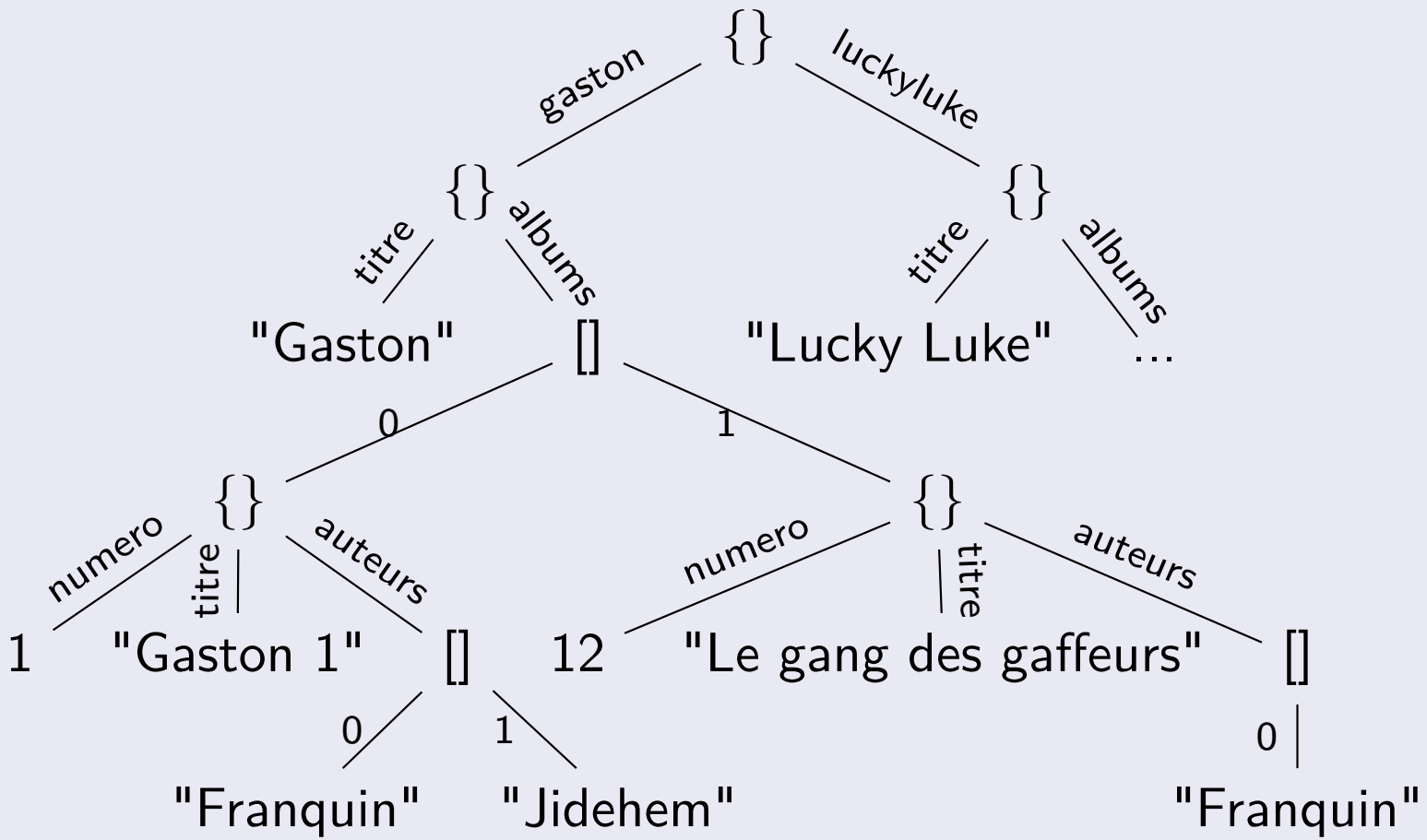
Exemple de document JSON

Collection d'albums de BD

```
{ "gaston": { "titre": "Gaston",
              "albums": [
                { "numero": 1,
                  "titre": "Gaston 1",
                  "auteurs": [ "Franquin", "Jidehem" ] },
                { "numero": 12,
                  "titre": "Le gang des gaffeurs",
                  "auteurs": [ "Franquin" ]}
              ] },
  "luckyluke": { "titre": "Lucky Luke",
                 "albums": [
                    { "numero": 1,
                      "titre": "La mine d'or de Dick Digger",
                      "auteurs": [ "Morris" ]}
                 ] } }
```

Exemple avec arbre

Extrait de la collection précédente



Chercher de l'information dans un arbre

Deux possibilités

- Programmatiquement dans un langage générique, *e.g.* en Python ou en Javascript
- En utilisant un langage dédié, plus compact : JSONPath et SQL/JSONPath

Approche programmatique

Parfois compacte pour des recherches simples :

Recherche en JS

```
data = ...
data.gaston.albums[0].auteurs[0]
```

Le premier auteur du premier album de Gaston

Approche programmatique

Parfois compacte pour des recherches simples :

Recherche en Python

```
data = ...
data["gaston"]["albums"][0]["auteurs"][0]
```

Le premier auteur du premier album de Gaston

Approche programmatique : plusieurs valeurs

Extraction simple en JS :

Extraire le titre de chaque série

impératif

```
ex2 = [];
for (s in data) {
    ex2.push(data[s].titre);
}
```

fonctionnel

```
ex2 = Object.values(data).map((s) => s.titre);
```

Approche programmatique : plusieurs valeurs - suite

Extraction simple en JS :

*Extraire le titre de chaque **album***

impératif

```
ex3 = [];
for (s in data) {
    for (a of data[s].albums) {
        ex3.push(a.titre);
    }
}
```

Approche programmatique : plusieurs valeurs - suite

Extraction simple en JS :

*Extraire le titre de chaque **album***

fonctionnel

```
ex3 = Object.values(data)
      .flatMap((s) => s.albums)
      .map((a) => a.titre);
```

Approche programmatique : filtre sur valeur

Extraction avec filtre en JS :

Extraire le titre des séries
dont on possède l'album numéro 10 ou plus

impératif

```
ex4 = [];
for (s in data) {
    for (a of data[s].albums) {
        if (a.numero >= 10) {
            ex4.push(data[s].titre);
            break;
        }
    }
}
```

Approche programmatique : filtre sur valeur

Extraction avec filtre en JS :

Extraire le titre des séries
dont on possède l'album numéro 10 ou plus

fonctionnel

```
ex4 = Object.values(data)
    .filter((s) =>
        s.albums.some(
            (a) => a.numero >= 10))
    .map((s) => s.titre);
```

Langage dédié : JSONPath

Langage de requêtes

- Pour chercher de l'information dans des documents JSON
- ou comme critère de filtrage de documents JSON

Langage de chemins :

- Expression = spécification de chemin dans un arbre JSON
- Sémantique :

$$\text{expression} + \text{nœud de départ} \rightsquigarrow \text{ensemble de nœuds}$$

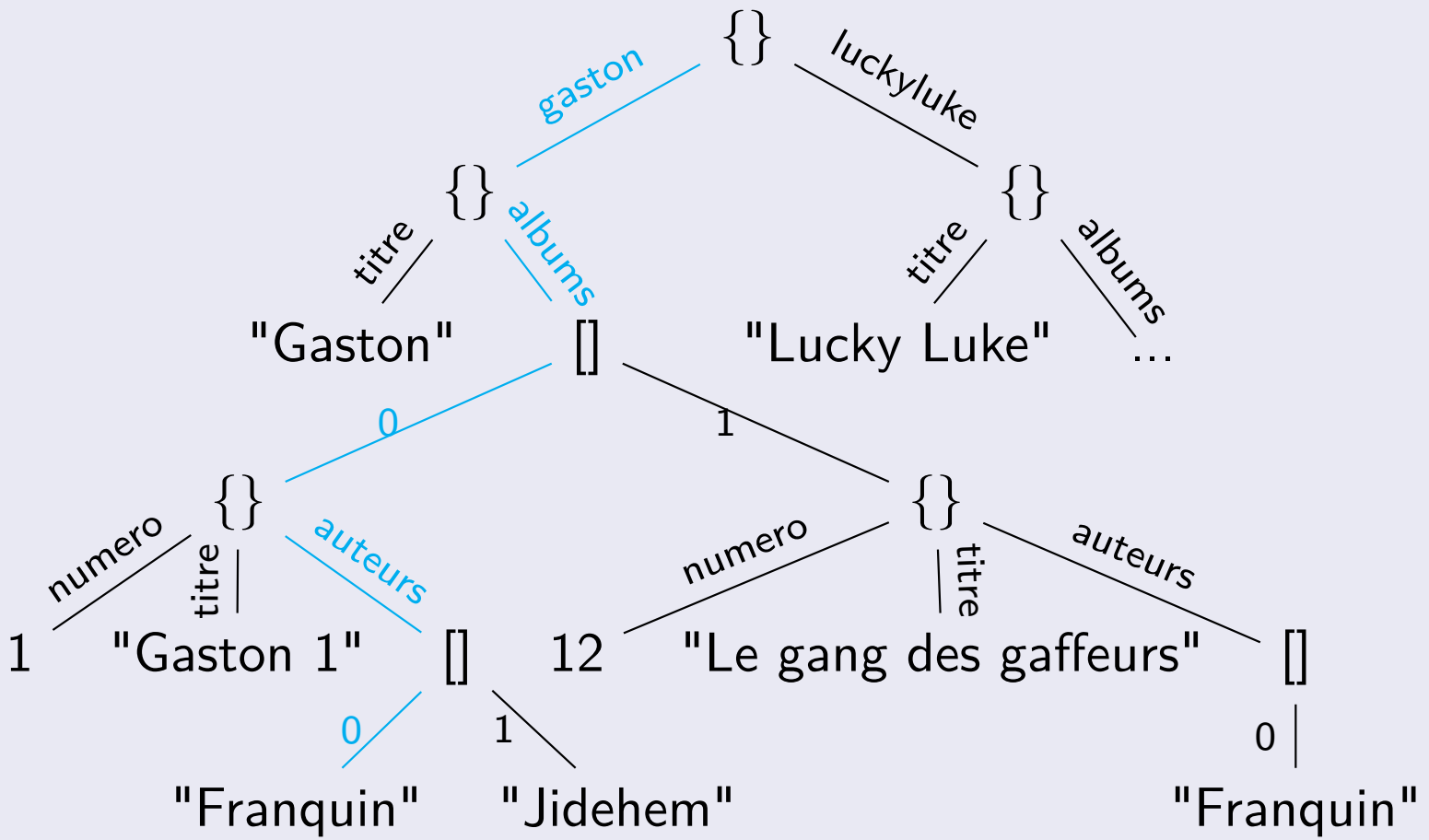
Langages de chemins : analogie avec les chemins de fichier dans un shell bash

Noms de fichier dans les commandes shell :

- nom absolu, e.g. `/home/ecoquery/bdnosql/cm1.tex`
  $\rightarrow$ un seul fichier possible
- nom relatif, e.g. `bdnosql/cm1.tex`
  $\rightarrow$ un seul fichier,n mais qui dépend du répertoire de travail
- avec *wildcard*, e.g. `bdnosql/*.tex`
  $\rightarrow$ plusieurs fichiers possibles
  (plusieurs chemins possibles, un par fichier)
- avec plusieurs *wildcards*, e.g. `*/*.tex`

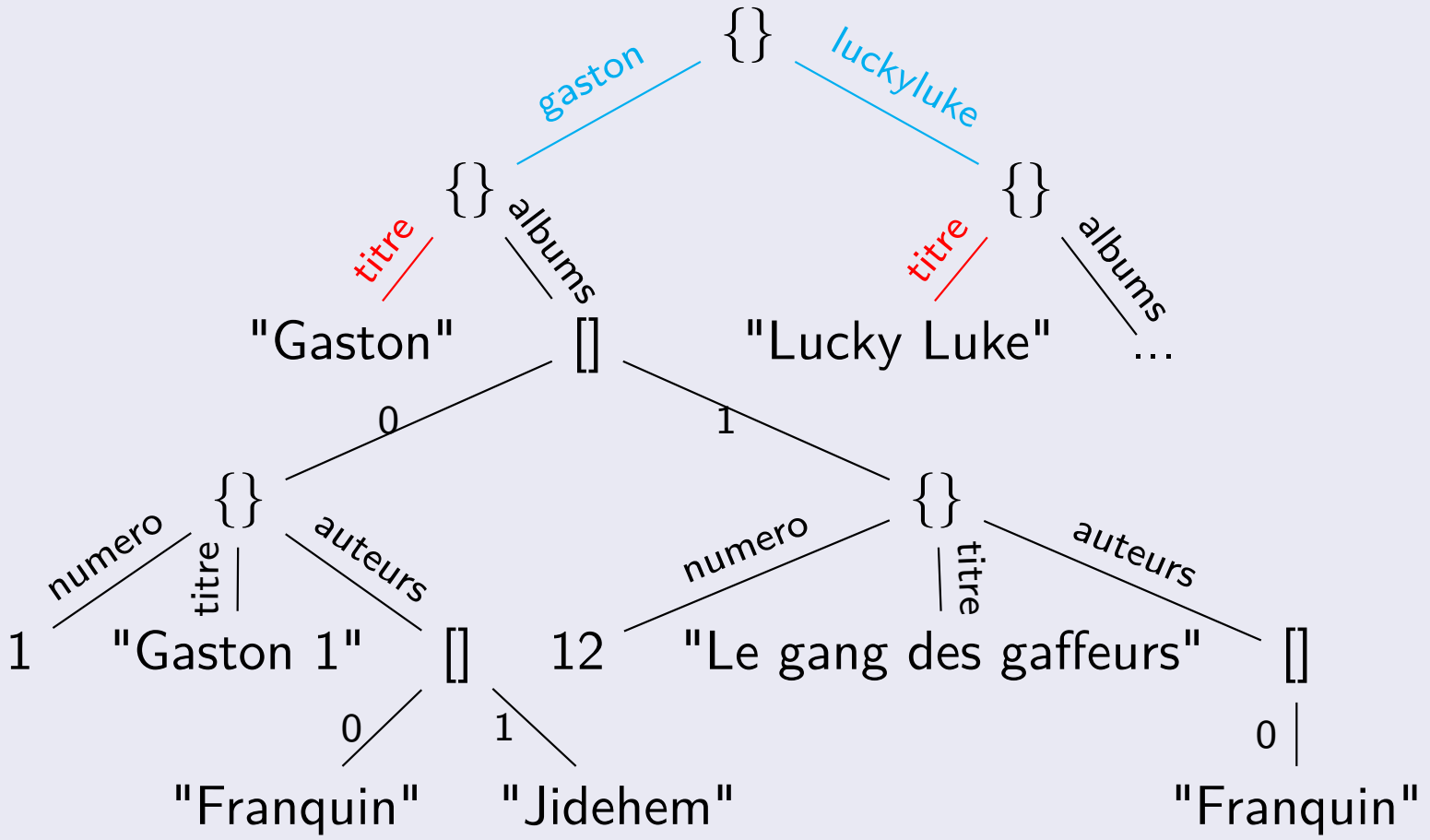
Exemple JSONPath : Le premier auteur du premier album de la série Gaston

```
$.gaston.albums[0].auteurs[0]
```



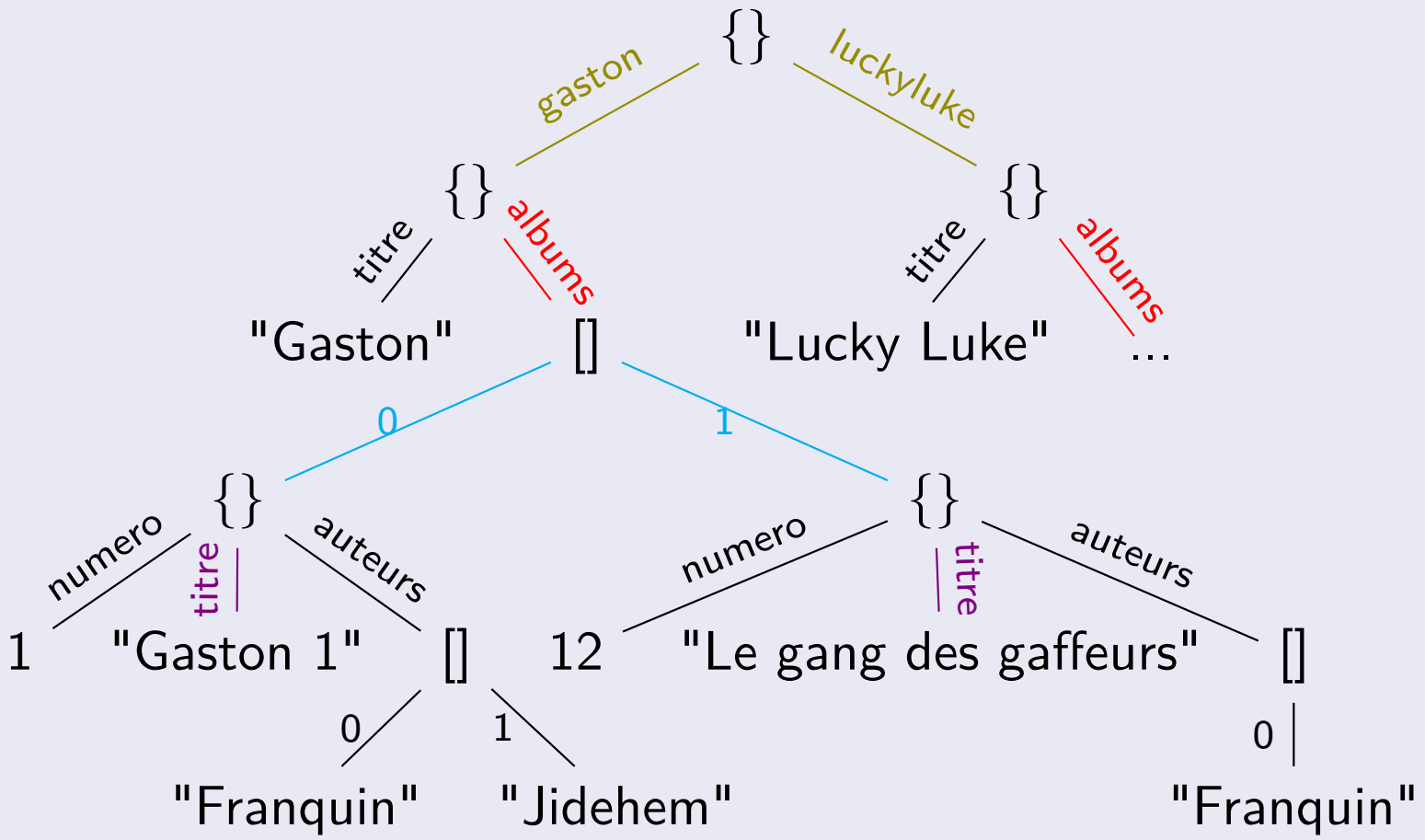
Exemple JSONPath : le titre de chaque série

```
$.*.titre
```



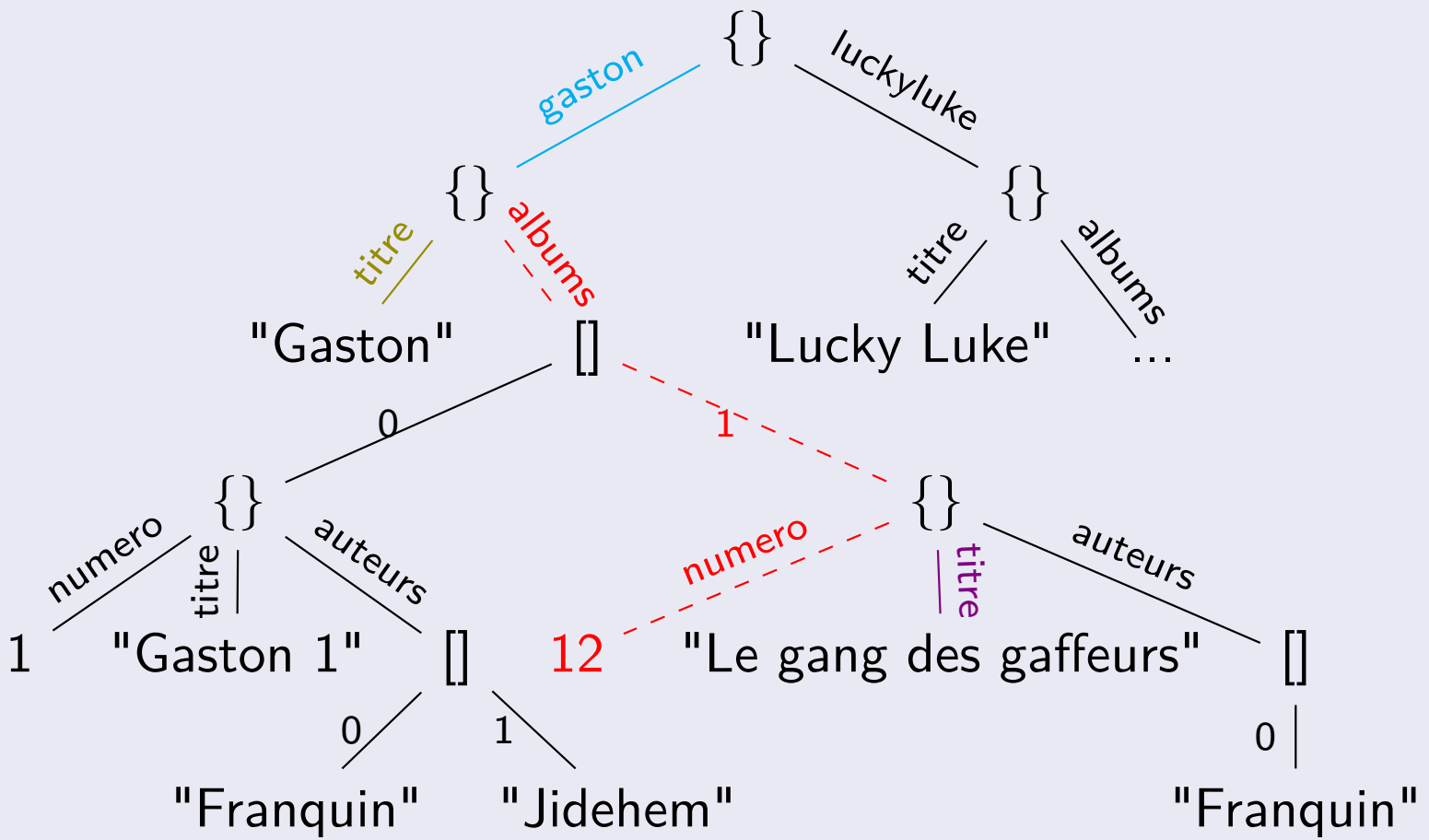
Exemple JSONPath : le titre de chaque album

```
$.*.albums[*].titre
```



Exemple JSONPath : le titre des séries dont on possède l'album numéro 10 ou plus

```
$.* ?(@.albums[*].numero >= 10) .titre
```



Syntaxe JSONPath (extrait) chemins

exprC	::=	`$`	racine
	|	`@`	nœud courant
	|	*exprC* `.`*champ*	valeur du champ
	|	*exprC* `.*`	valeur de tous les champs
	|	*exprC* `..*`	valeur de tous les champs, récurs.
	|	*exprC* `.**`	idem, mais en SQL/JSON Path
	|	*exprC* `[`*entier*`]`	case tableau
	|	*exprC* `[*]`	toutes les valeurs d'un tableau
	|	*exprC*`?(`*exprB*`)`	filtre sur cond. bool. ($\approx$ WHERE)

Syntaxe JSONPath (extrait) conditions

exprB	::=	`true`	
	|	`false`	
	|	`!` *exprB*	négation
	|	*exprB* `&&` *exprB*	et logique
	|	*exprB* `||` *exprB*	ou logique
	|	*exprC op val*	comparaison avec valeur
	|	*exprC op exprC*	comparaison avec chemin
	|	`exists(`*exprC*`)`	test d'existence d'un chemin relatif

avec $op \in \{==, <>, !=, <, <=, >, >=\}$
et *exprC* commence par `@` ou `$`

Opérations sur des documents JSON

On suppose prédéfinies les opérations suivantes sur les documents JSON (notés j) :

- *acces*(j, x) : accès au champ ou à l'indice x de l'objet / du tableau j. Renvoie un singleton avec cette valeur ou un ensemble vide si elle n'est pas définie.
  en JS : `j[x]!==undefined ? [j[x]] : []`
- *valeurs*(j) valeurs de l'objet / du tableau.
  en JS : `Array.from(j.values())`

Opération d'extraction des sous-documents

sousDoc récupère tous les sous-documents de son argument

$$sousDoc(x) = \begin{cases} \{x\} & \text{si } x \text{ est atomique} \\ \{x\} \cup \bigcup_{y \in valeurs(x)} sousDoc(y) & \text{sinon} \end{cases}$$

Interprétation d'une expression JSONPath

Sémantique : fonction *eval* : $JSON \times JSON \times JSONPath \rightarrow 2^{JSON}$.
Dans la suite

- *exprC* est une expression de chemin JSONPath
- *exprB* est une expression de condition JSONPath
- *r* est la racine du document JSON que l'on requête
- *c* est le sous-document JSON de contexte (lorsque cela a du sens)
- *j* est un (sous) document JSON

Définition de *eval*

$$eval(r, c, \$) = \{r\}$$
$$eval(r, c, @) = \{c\}$$
$$eval(r, c, exprC.champ) = \bigcup_{j \in eval(r,c,exprC)} acces(j, champ)$$
$$eval(r, c, exprC.*) = \bigcup_{j \in eval(r,c,exprC)} valeurs(j)$$
$$eval(r, c, exprC[n]) = eval(r, c, exprC.n)$$
$$eval(r, c, exprC[*]) = eval(r, c, exprC.*)$$
$$eval(r, c, exprC..*) = \bigcup_{j \in eval(r,c,exprC)} sousDoc(j)$$
$$eval(r, c, exprC?(exprB)) = \{j \in eval(r, c, exprC) \mid (r, j) \models exprB\}$$

Interprétation des comparaisons

exprC op val

Évaluer *exprC* en prenant pour @ le nœud à tester :

- conserver le nœud à tester uniquement si on peut trouver un *r* parmi les résultats tels que *r op val* est vrai

Rmq : cela suppose que l'on sait évaluer *r op val*, ce qui est le cas si *r* est atomique.

Interprétation des conditions : définition de $\models$

r (racine) et c (contexte) sont des nœud d'arbre JSON.

$(r, c) \models \texttt{true}$

$(r, c) \models \texttt{!}\mathit{exprB}$ si $(r, c) \not\models \mathit{exprB}$

$(r, c) \models \mathit{exprB}_1 \texttt{ \&\& } \mathit{exprB}_2$ si $(r, c) \models \mathit{exprB}_1$ et $(r, c) \models \mathit{exprB}_2$

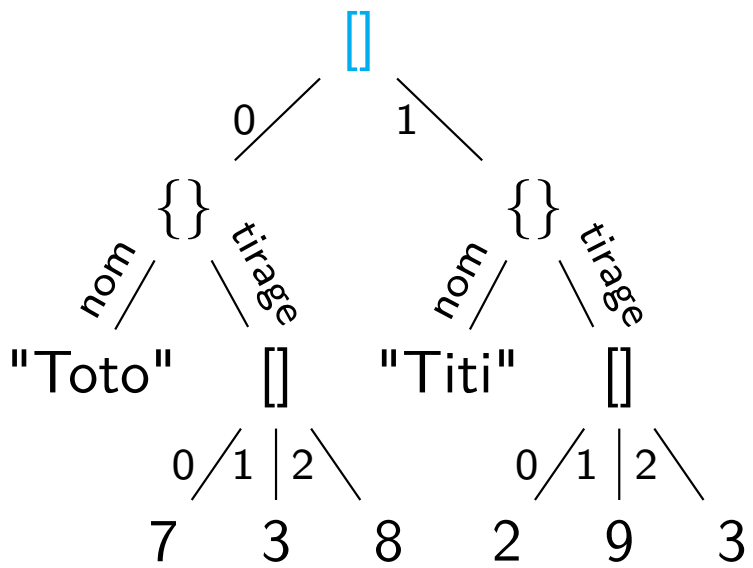
$(r, c) \models \mathit{exprB}_1 \texttt{ || } \mathit{exprB}_2$ si $(r, c) \models \mathit{exprB}_1$ ou $(r, c) \models \mathit{exprB}_2$

$(r, c) \models \mathit{exprC}\ \mathit{op}\ \mathit{val}$ si $\exists j \in \mathit{eval}(r, c, \mathit{exprC})$ tel que $j\ \mathit{op}\ \mathit{val}$ est vrai

$(r, c) \models \mathit{exprC}_1\ \mathit{op}\ \mathit{exprC}_2$ si $\exists j_1 \in \mathit{eval}(r, c, \mathit{exprC}_1), \exists j_2 \in \mathit{eval}(r, c, \mathit{exprC}_2)$ tels que $j_1\ \mathit{op}\ j_2$ est vrai

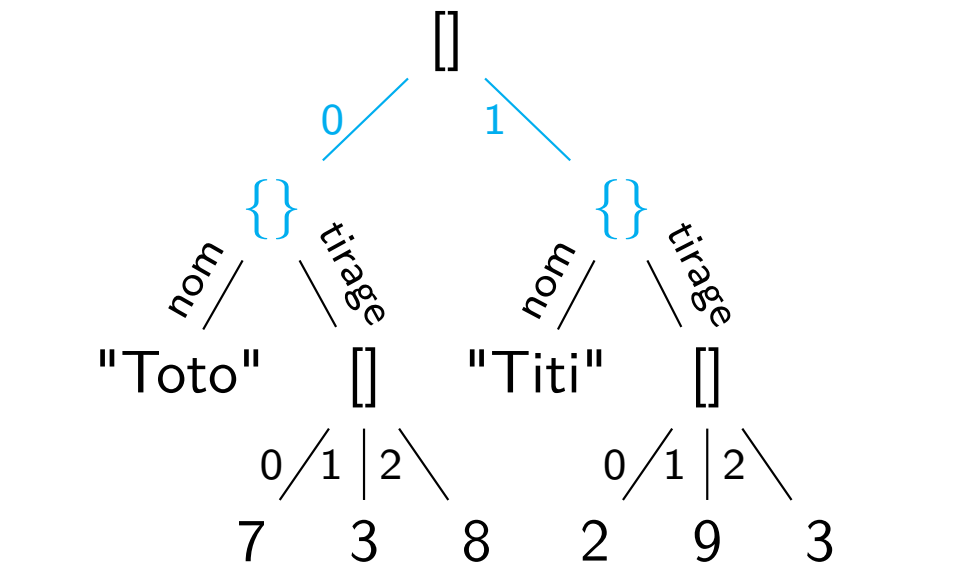
$(r, c) \models \texttt{exists(}\mathit{exprC}\texttt{)}$ si $\mathit{eval}(r, c, \mathit{exprC}) \neq \emptyset$

Comparaison : exemple



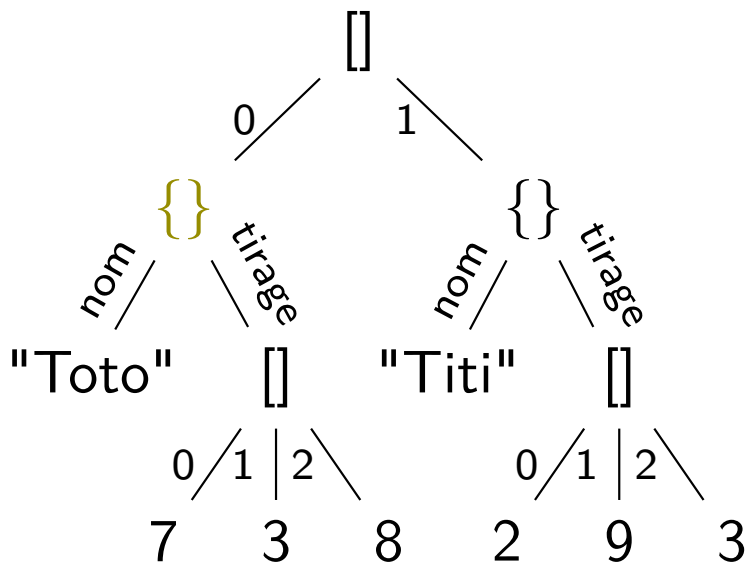

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



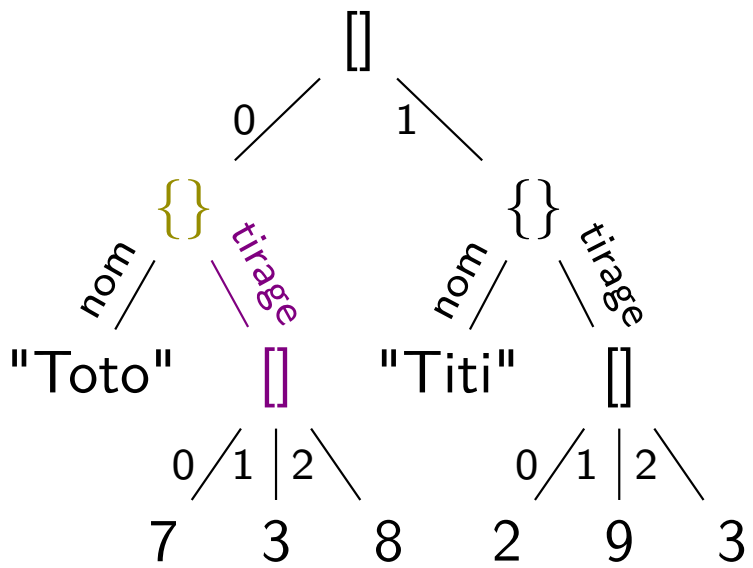

```
$[*] ?(@.tirage[*] == 9)  .nom
```

Comparaison : exemple



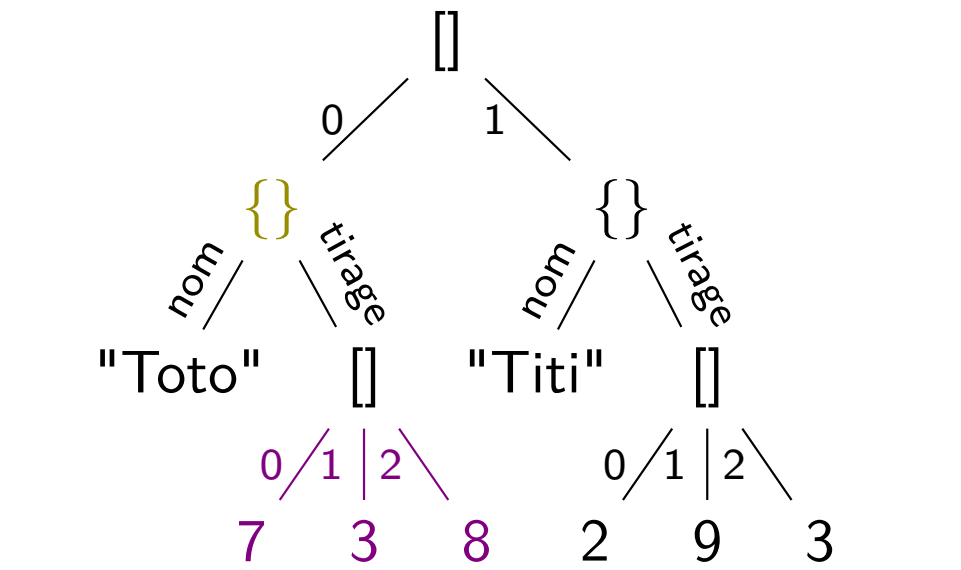

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



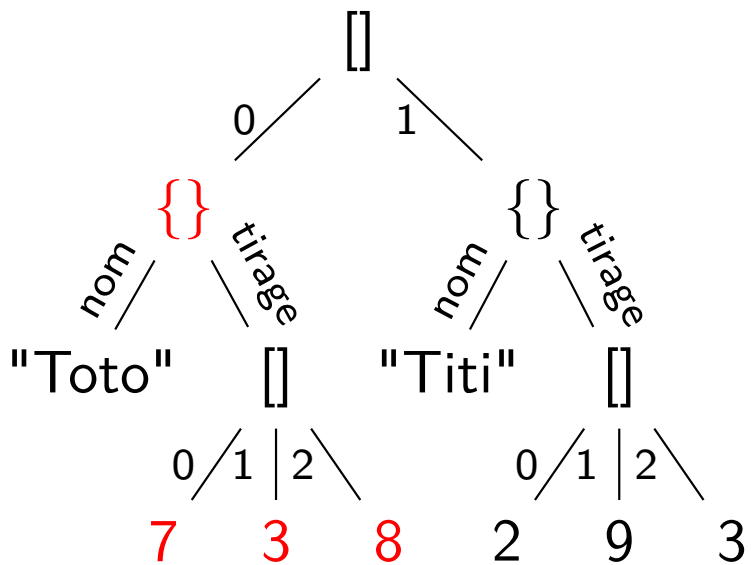

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



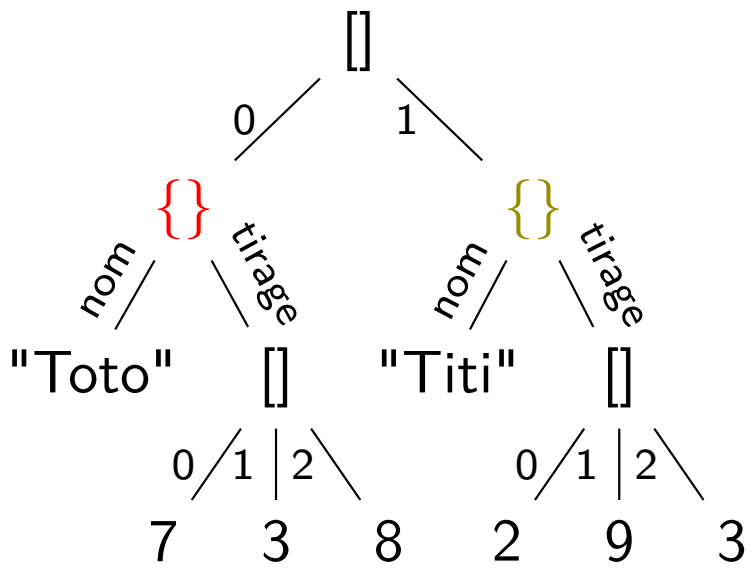

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



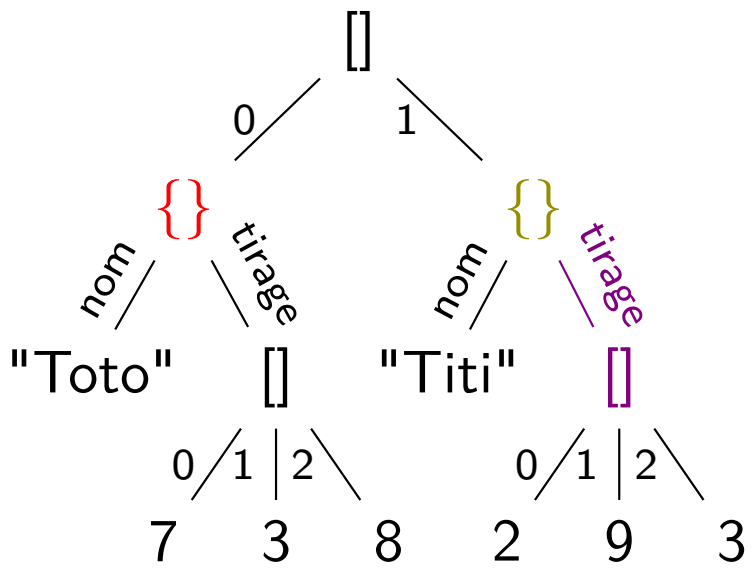

```
$[*] ?(@.tirage[*] == 9)  .nom
```

Comparaison : exemple



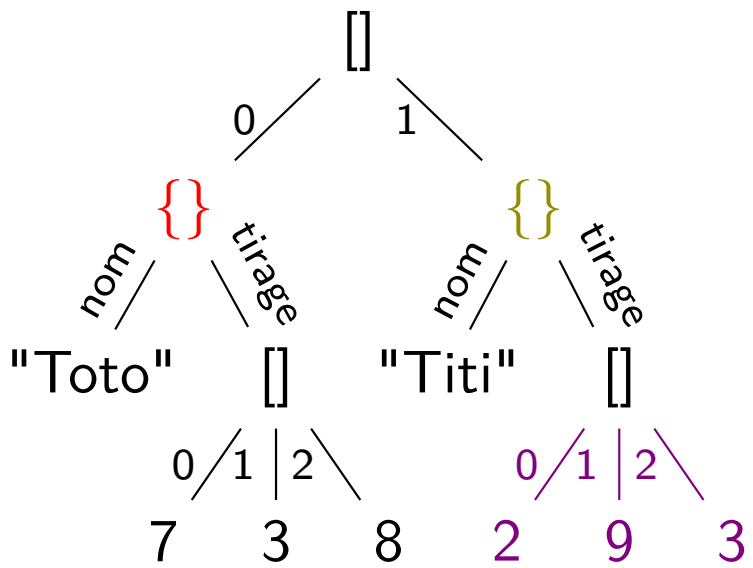

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



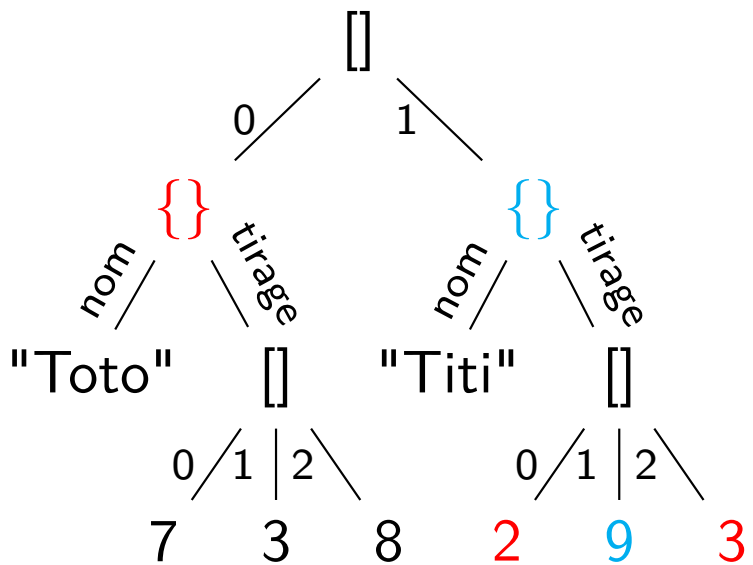

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple



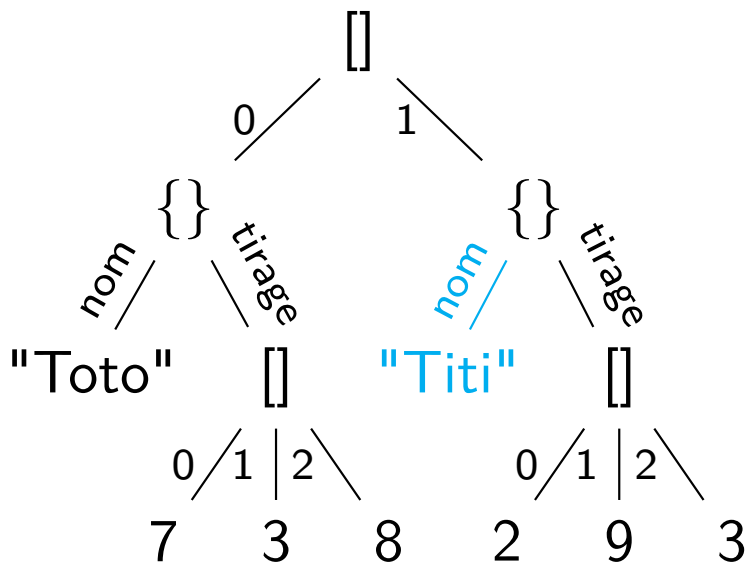

```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple




```
$[*] ?(@.tirage[*] == 9)   .nom
```

Comparaison : exemple




```
$[*] ?(@.tirage[*] == 9)   .nom
```

Alternatives : jq et json natif dans PostgreSQL

JSONPath	jq	PostgreSQL
ex.*champ*	*ex* | .*champ*	*ex* -> *champ*
ex.*	*ex* | .[]	json_each
ex..*champ*	*ex* | ..	(requête récursive)
ex[*n*]	*ex* | .[*n*]	*ex* -> *n*
ex[*]	*ex* | .[]	json_array_elements
ex ?(*eb*)	*ex* | select(*eb*)	sous-requêtes + WHERE

Rmq : JSONPath dans PostgreSQL via @? et jsonb_path_query

Références

- https://goessner.net/articles/JsonPath/
- https://golangexample.com/abstract-json-for-golang-with-jsonpath-support/
- https://www.postgresql.org/docs/current/functions-json.html
- https://stedolan.github.io/jq/