

Rapport : Projet NTIC

Interface de création d'arbres syntaxiques

Présentation :

Le but de ce projet est de concevoir une interface Web permettant de créer facilement des arbres syntaxiques.

Cette interface a 2 utilisateurs-type :

- Les linguistes / enseignants de linguistique
- Les étudiants de linguistique

L'idée de créer une telle interface est partie du constat que les outils existants pour la syntaxe laissent à désirer. Soit ils sont trop compliqués et la création d'arbres prend trop longtemps (ex: *ArborWin*¹); soit les arbres générés sont statiques et ne peuvent pas être modifiés (ex: *TreeForm*²).

Il est donc essentiel que ce projet permette de créer des arbres d'une **façon simple et intuitive**, et que l'outil soit pensé pour **être utilisé pendant des cours** de syntaxe (démonstration de création d'arbre aux étudiants, via *Zoom* par ex.).

Mockup de l'interface visuelle :

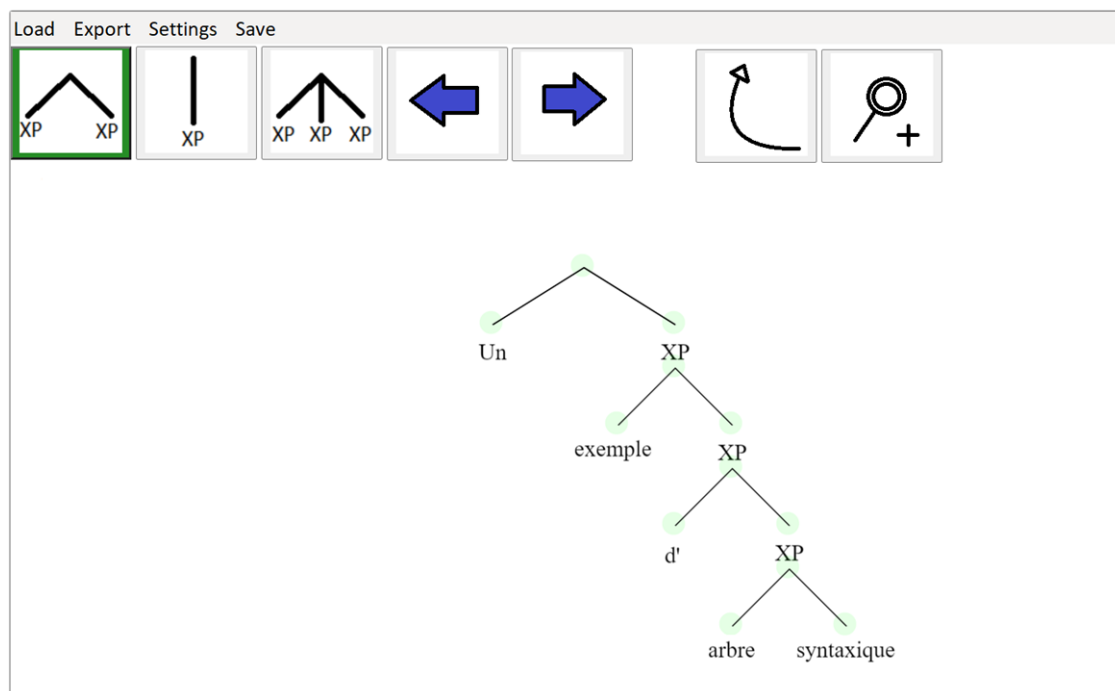


Figure 1: mockup de l'interface utilisateur

¹ <https://www.cascadilla.com/arboreal.html>

² <https://www.mapsofspeech.com/treeform/>

Plusieurs fonctionnalités sont aussi prévues par un click-droit sur l'arbre : rajouter un noeud; supprimer un noeud; modifier un noeud (couleurs, nombre de bras).

Cahier des charges initial :

- Implémenter la structure de base de l'interface : un canvas HTML supportant un modèle-objet pour créer et éditer des noeuds (binaires ou +) d'arbre.
- Implémenter l'algorithme pour résoudre les intersections de noeuds dans l'interface.
- Rajouter les fonctionnalités de bases pour un "MVP" (*produit minimum viable*) : flèches de mouvement syntaxique, modification de noeuds par *drag&drop*, Undo/Redo, *zoom*, triangles syntaxiques pour "cacher" sous-arbre.
- Implémenter les fonctions d'exportation (en format SVG) et importation d'arbres (depuis un fichier JSON). L'exportation doit permettre d'intégrer un arbre facilement à un document Word/PDF.

Cas d'utilisation :

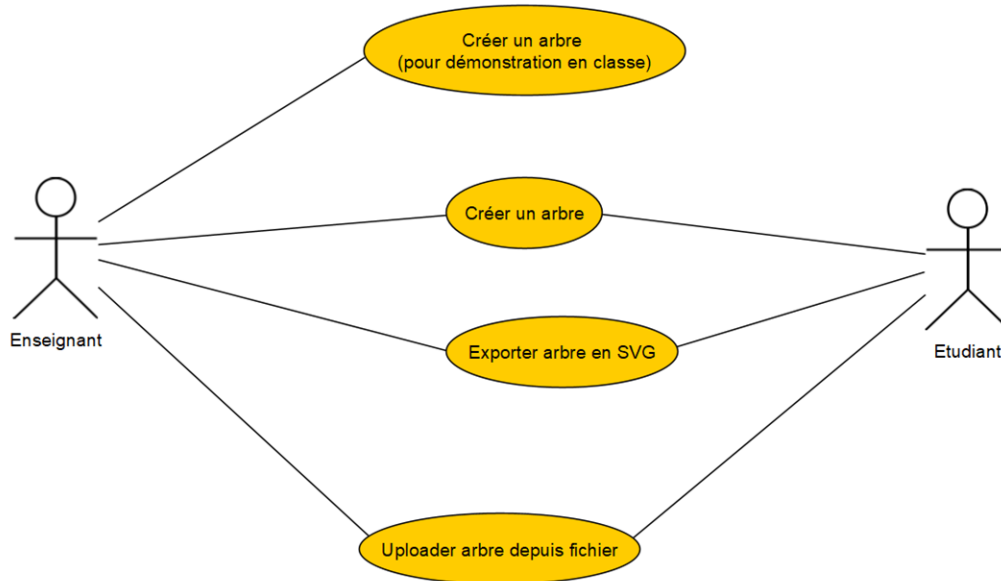


Figure 2: cas d'utilisation de l'outil

Le diagramme ci-dessus vise à mettre en valeur le fait que, dans la salle de classe, un enseignant a des besoins différents de l'outil en question. Par exemple, on peut vouloir *zoomer* sur une partie spécifique d'un arbre, ou bien mettre en valeur certains éléments avec des couleurs pour attirer l'attention des étudiants.

Présentation de l'interface:

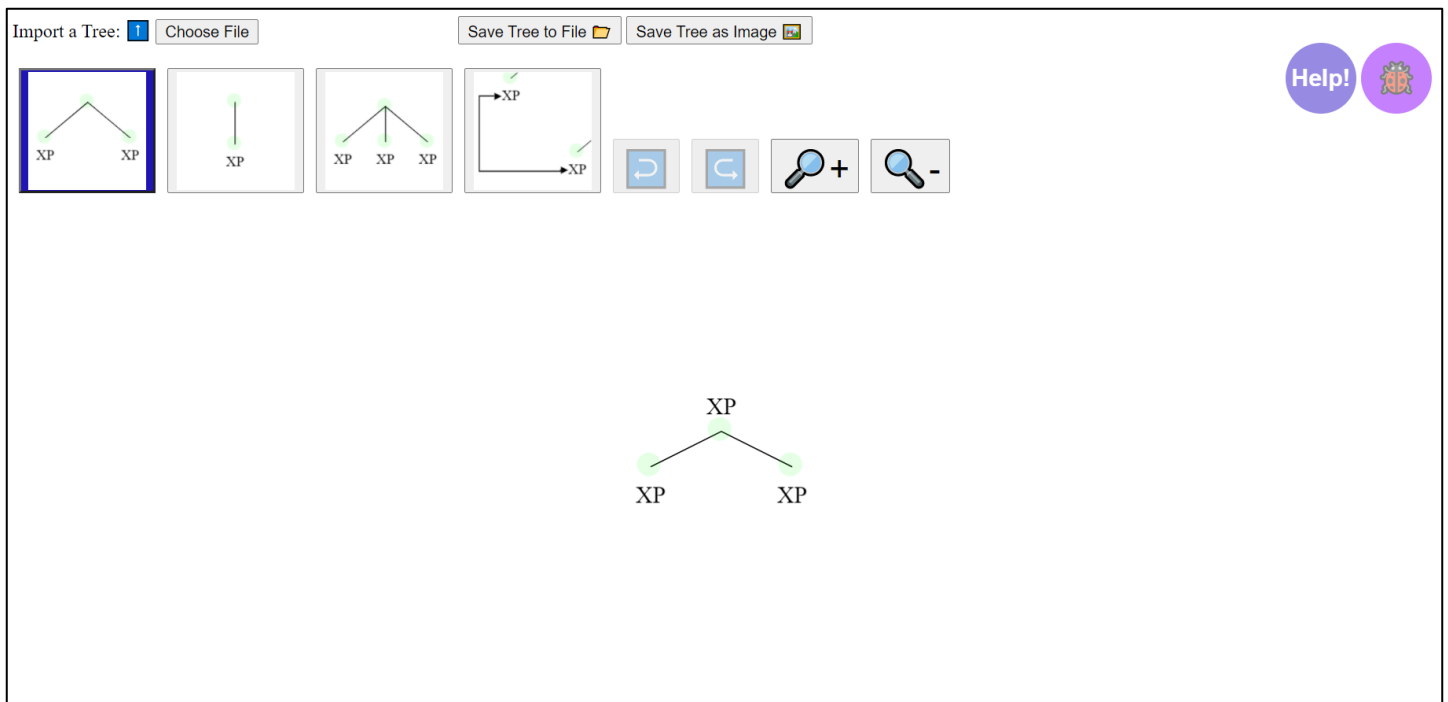


Figure 3: l'interface au terme du projet

L'interface obtenue (lien temporaire [ici](https://nifty-morse-4f2438.netlify.app/)³) au bout du projet est finalement très similaire au *mockup* de départ. La seule véritable différence est l'ajout de deux boutons sur le coin droit de l'écran : un bouton 'Help!' qui mène l'utilisateur vers un guide d'utilisation, et un bouton destiné à soumettre des rapports de *Bug*.

Un guide utilisateur⁴ a été rajouté pour simplifier la prise en main de l'outil, avec des GIFs qui démontrent les différentes manipulations possibles.

Conception du Programme:

Le diagramme UML ci-dessous représente les principales classes du projet.

Le canevas utilisé par l'utilisateur pour créer les arbres n'est rien de plus qu'un élément HTML `<canvas>`. Cependant, ce `<canvas>` est totalement retravaillé par le framework **Fabric.js**⁵, qui lui rajoute tout un modèle-objet pour le manipuler facilement. La présence d'un tel modèle-objet était essentielle pour permettre la manipulation des arbres.

Toute la logique du canevas a été codée en JavaScript, langage utilisé par Fabric.js et parfaitement adapté au développement web. Le plugin **jQuery contextMenu**⁶ a été utilisé pour rajouter les fonctionnalités de click-droit facilement.

³ <https://nifty-morse-4f2438.netlify.app/>

⁴ <https://nifty-morse-4f2438.netlify.app/tutorial.html>

⁵ <https://fabricjs.com/>

⁶ <https://swisnl.github.io/jQuery-contextMenu/>

Ce **fabric.Canvas** est associé à la classe **CanvasHistory**, qui permet d'implémenter la fonctionnalité *Undo/Redo* en gardant un Stack de toutes les actions faites par l'Utilisateur.

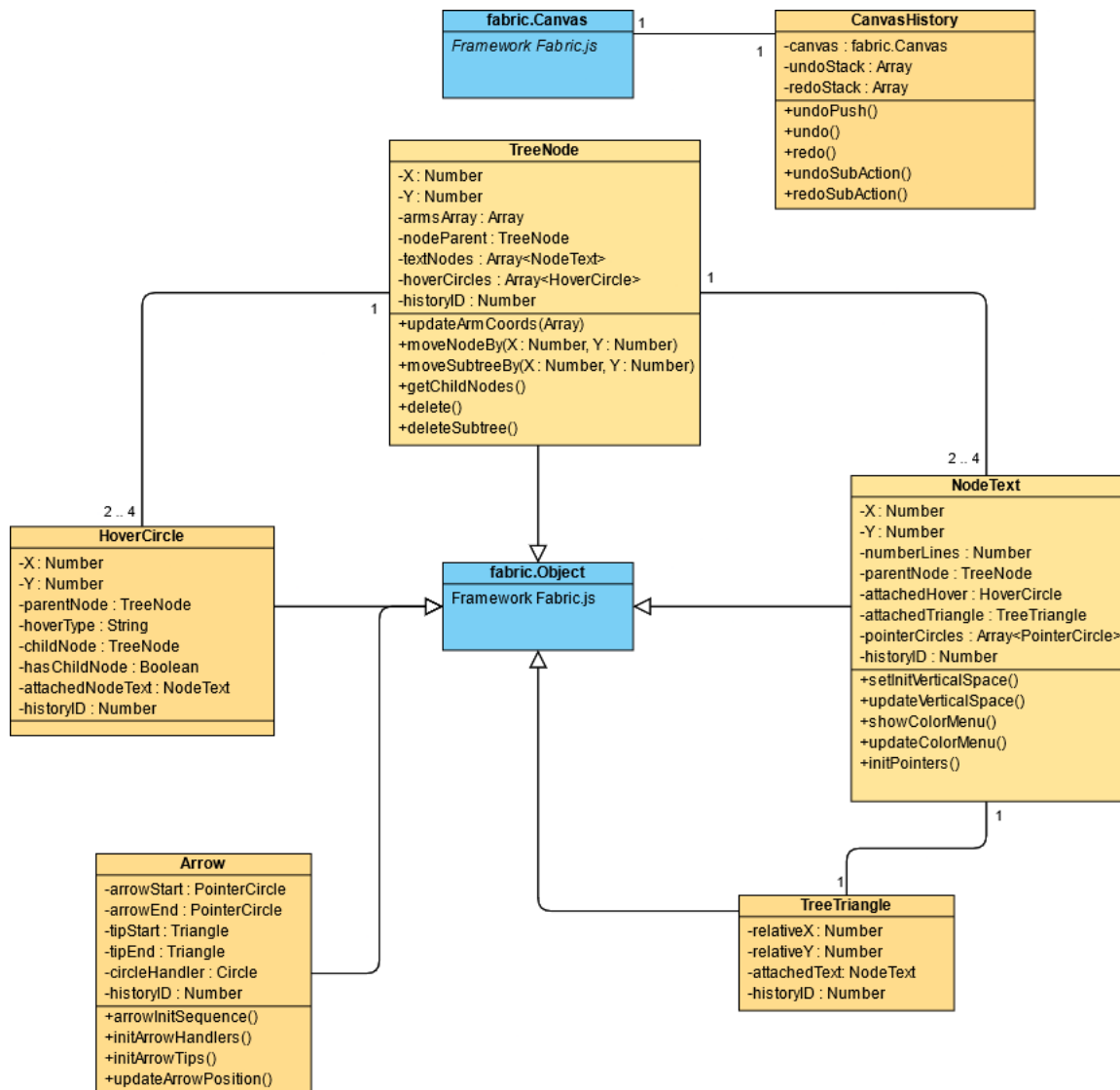


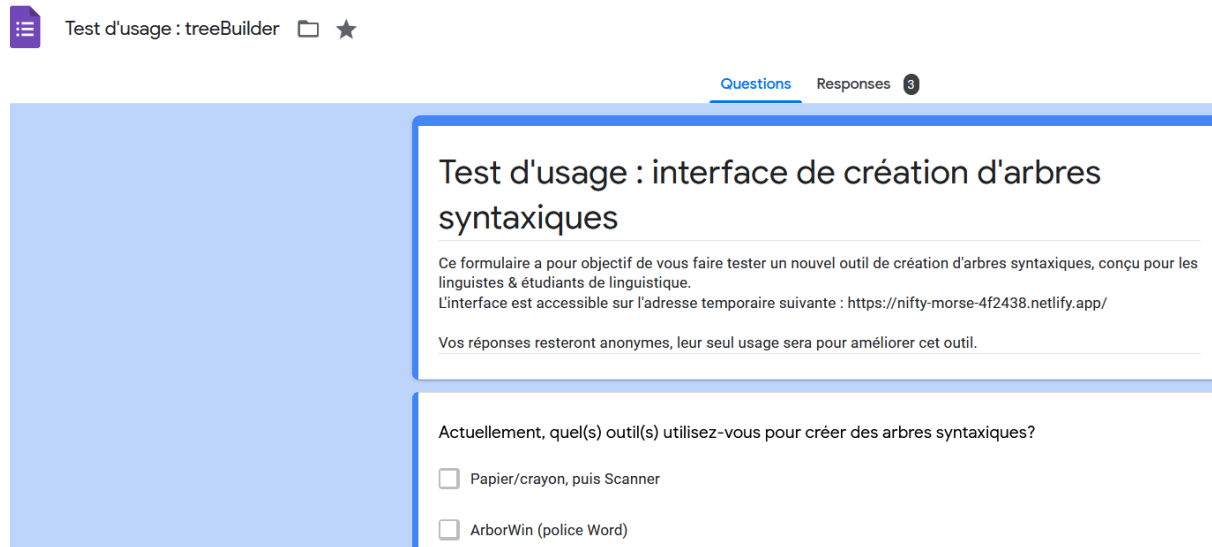
Figure 4: diagramme des classes UML

Toutes les autres classes (**TreeNode**, **HoverCircle**, etc...) sont des sous-classes de **fabric.Object**. Cette dernière permet d'associer un objet à une variable, de lui assigner des propriétés, et de le bouger dans le canevas sans se préoccuper d'effacer son ancienne position.

Un objet **TreeNode** représente une branche d'arbre. Chaque branche a des objets **HoverCircle** sur ces extrémités, qui permettent de modifier la forme et position de la branche. Les extrémités de chaque branche sont aussi associées à un **NodeText**, la case texte qui permet de nommer chaque bras et indiquer les [features] associées à ce bras. Les **TreeTriangle** associés à **NodeText** symbolisent un sous-arbre qui n'est pas représenté, ce qui permet de simplifier des arbres très complexes par différents degrés d'abstraction.

Tests Effectués:

Avec un prototype qui marche suffisamment bien, il était essentiel de tester l'interface avec de potentiels utilisateurs.



The screenshot shows a Google Forms interface. At the top, there's a tab for 'Questions' and another for 'Responses' with a count of 3. The main title of the form is 'Test d'usage : interface de création d'arbres syntaxiques'. Below the title, there's a description: 'Ce formulaire a pour objectif de vous faire tester un nouvel outil de création d'arbres syntaxiques, conçu pour les linguistes & étudiants de linguistique. L'interface est accessible sur l'adresse temporaire suivante : <https://nifty-morse-4f2438.netlify.app/>'. A note states: 'Vos réponses resteront anonymes, leur seul usage sera pour améliorer cet outil.' The first question is 'Actuellement, quel(s) outil(s) utilisez-vous pour créer des arbres syntaxiques?'. It has two radio button options: 'Papier/crayon, puis Scanner' and 'ArborWin (police Word)'. Both options are currently unselected.

Figure 5: questionnaire en ligne pour le test de l'interface

Un questionnaire en ligne⁷ a été conçu pour cela : ce dernier demande à l'utilisateur de se familiariser avec le guide d'utilisation de l'interface, puis ensuite d'essayer de créer 3 arbres syntaxiques différents, et de tester les fonctionnalités d'export. À chaque étape, l'utilisateur peut noter les difficultés rencontrées, et à la fin du questionnaire des suggestions d'amélioration sont demandées.

Malgré un nombre faible de répondants (N=3, étudiants de linguistique & linguistes), les réponses ont été très détaillées. Par ailleurs, un nombre d'internautes ont testé l'interface et laissé des *feedbacks* de façon anonyme.

De tous ces retours, nous retenons :

Bugs de l'interface:

- problèmes de taille de l'export SVG avec certains arbres.
- click-droit ne fonctionne pas correctement parfois.
- gestion des flèches est encore maladroite.
- manque d'outils d'édition texte--italique, exposants, etc...

Suggestions d'amélioration:

- Outil pour attirer l'attention sur une partie d'un arbre (cercles/rectangles superposés)
- Possibilité d'importer/exporter des arbres représentés sous forme de crochets imbriqués (ex: [\[S \[NP This\] \[VP \[V is\] \[^NP a wug\]\]\]](#))
- Un système de gestion des traits interprétables/ininterprétables permettant de faire la vérification de traits.
- Plus de raccourcis clavier (ctrl+Z, ctrl+Y).

⁷ <https://forms.gle/rDiKXxNrj6mMQ7Nb8>

Développements futurs:

Bien que le projet satisfait déjà les points définis initialement sur le cahier des charges (*bugs* à part), plusieurs améliorations sont prévues :

- Rajout des suggestions données par *feedback* (cf. section *Tests* ci-dessus).
- Ajout de flèches courbées, et possibilité de modifier les deux extrémités des flèches représentant les mouvements syntaxiques.
- Ajout d'un bouton '*Custom Branch*' permettant à l'Utilisateur de sauvegarder un type basique de branche avec les propriétés de son choix (nombre et longueur des bras, contenu et couleur par défaut des cases texte, etc...).

Zoom sur le code:

Une des fonctionnalités les plus intéressantes implémentées pour ce projet est la sauvegarde d'un arbre sous format JSON .

```

367 function flattenObjects(array) {
368     array.forEach(function (object) {
369
370         if (object.type == 'hoverCircle') {
371             object.parentNode = (object.parentNode == null ? null : object.parentNode.historyID);
372             object.childNode = (object.childNode == null ? null : object.childNode.historyID);
373             object.attachedNodeText = (object.attachedNodeText == null ? null : object.attachedNodeText.historyID);
374         }
375         else if (object.type == 'treeNode') {
376             object.nodeParent = (object.nodeParent == null ? null : object.nodeParent.historyID);
377             object.hoverParent = (object.hoverParent == null ? null : object.hoverParent.historyID);
378             object.topTextNode = (object.topTextNode == null ? null : object.topTextNode.historyID);
379
380             object.hoverCircles.forEach(function (item, innerIndex) {
381                 object.hoverCircles[innerIndex] = object.hoverCircles[innerIndex].historyID;
382             });
383
384             object.textNodes.forEach(function (item, innerIndex) {
385                 object.textNodes[innerIndex] = object.textNodes[innerIndex].historyID;
386             });
387         }
388         else if (object.type == 'nodeText') {
389             object.parentNode = (object.parentNode == null ? null : object.parentNode.historyID);
390             object.attachedHover = object.attachedHover.historyID;
391             object.mainTextNode = (object.mainTextNode == null ? null : object.mainTextNode.historyID);
392             object.secondaryText = (object.secondaryText == null ? null : object.secondaryText.historyID);
393             object.attachedTriangle = (object.attachedTriangle == null ? null : object.attachedTriangle.historyID);
394         }
395     });
396 }

```

Figure 6: zoom sur le code 1

JavaScript possède les fonctions prédéfinies `JSON.stringify()` et `JSON.parse()` pour sérialiser et dé-sérialiser des objets. Cependant, il n'a pas été aussi simple que cela de sauvegarder tous les objets du canevas.

Le problème était que les objets contiennent parmi leurs attributs des références entre eux. Par ex., tout objet **TreeNode** contient un attribut **textNodes** qui est une liste de tous ses objets **NodeText**. Ces derniers contiennent à leur tour une référence à leur branche **TreeNode**.

Ces références circulaires sont essentielles pour le fonctionnement de l'interface, mais malheureusement JSON n'aime pas du tout la circularité. Il est impossible de sérialiser des objets avec des références circulaires!

Pour résoudre cela, j'ai dû créer une méthode pour 'applatir' tous les objets du canevas. Cette méthode marche en remplaçant chaque attribut ayant une référence vers un autre objet, par un attribut de type Number représentant l'ID de l'objet.

```
if (object.type == 'hoverCircle') {
    object.parentNode = (object.parentNode == null ? null : object.parentNode.historyID);
}
```

Figure 7: zoom sur le code 2

Par exemple, pour tous les objets **HoverCircle** leur attribut **parentNode** est remplacé par **parentNode.historyID** (où bien **null** si l'objet n'a pas de **parentNode**).

Bien sûr, cela a demandé de rajouter un identifiant à chaque objet. Le système de **historyID** s'assure aussi que, en rechargeant un arbre depuis un fichier .JSON, tout nouveau objet rajouté au canevas par la suite n'est pas assigné un ID déjà utilisé.

Enfin, lorsqu'un arbre est importé vers l'application, il faut 'dé-applatir' chaque objet, en remplaçant tous les attributs aplatis par les références originelles. C'est ce que fait la fonction **reviveCanvasObject(o, object)**, qui appelle **findObjByID()** pour chaque attribut à récupérer :

```
591 function findObjByID(objectID, objectType) {
592     // no need to iterate if property is null
593     if (objectID != null) {
594         let objs = canvas.getObjects(objectType);
595         let result = null;
596
597         objs.forEach(function (object) {
598             if (object.historyID == objectID) {
599                 // console.log('Object found');
600                 result = object;
601             }
602         });
603         if (result == null) console.error(`Obj ${objectID}, type ${objectType} not found by ID`);
604         return result;
605     }
606     else { return null; }
607 }
```


Figure 8: zoom sur le code 3

Conclusion:

Note personnelle :

Ayant mis beaucoup (trop) d'heures de travail dans ce projet, je suis très content du résultat!

Cela me réjouit d'avoir reçu plusieurs messages d'étudiants éprouvant le même besoin d'un outil plus performant pour la syntaxe, et qui sont heureux de finalement en avoir un. Je compte continuer le développement de *TreeBuilder* dans le futur jusqu'à obtenir un outil complet et 100% fonctionnel.

Une note sur le nom *TreeBuilder*: oui, ce n'est pas très créatif  Si vous lisez cela, n'hésitez pas à m'envoyer un mail pour proposer quelque chose de mieux.

Si vous êtes développeur : le code du projet est disponible sur GitHub⁸ sous une license GPLv3, ce qui signifie que vous pouvez y contribuer, mais aussi l'utiliser pour tout projet dérivé (du moment que vous partagez aussi le code de votre projet dérivé).

Enfin, un grand merci à tous ceux qui m'ont guidé et soutenu pour la construction de ce petit projet!

⁸ <https://github.com/Milou6/treeBuilderJS>