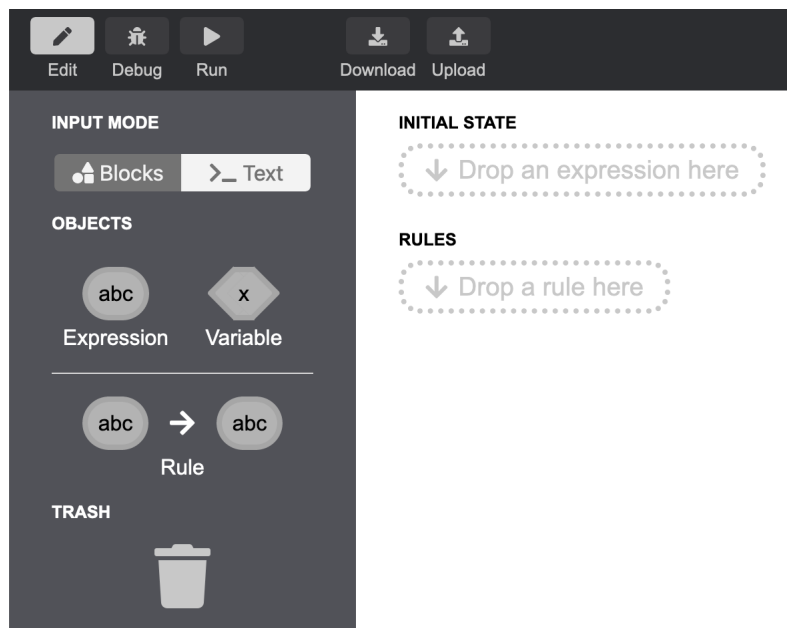




DÉPARTEMENT D'INFORMATIQUE  
UNIVERSITÉ DE GENÈVE  
TRAVAIL DE BACHELOR

# FUNVIEW

EXTENSION DE FUNBLOCKS  
GITHUB: [MATTHIEUVOS/FUNVIEW](#)



*Étudiant:*  
Matthieu VOS

*Professeur:*  
Didier BUCHS

SEMESTRE DE PRINTEMPS 2020

---

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contexte . . . . .	1
1.2	État de l’art . . . . .	2
<b>2</b>	<b>FunBlocks</b>	<b>5</b>
<b>3</b>	<b>Le module FunView</b>	<b>10</b>
3.1	La grammaire . . . . .	10
3.2	La sémantique . . . . .	13
3.3	Le parseur . . . . .	18
3.4	Le canvas HTML . . . . .	19
<b>4</b>	<b>Relier FunBlocks et FunView</b>	<b>22</b>
4.1	Ajouter FunView au projet FunBlocks . . . . .	22
4.2	Les appels à FunView . . . . .	23
<b>5</b>	<b>Conclusion</b>	<b>25</b>

# Chapitre 1: Introduction

---

## 1.1 Contexte

À l'heure actuelle, l'informatique est omniprésente. Qu'on l'utilise à des fins de loisirs ou de manière professionnelle, tout le monde utilise un outil informatique. Cependant, dans le système scolaire, l'informatique et son enseignement n'ont pas une très bonne image. Souvent relié aux mathématiques, beaucoup y voient un amas de formules. Cependant, l'informatique peut très bien être utilisée pour résoudre des problèmes sans calcul. Il suffit de voir tous les domaines dans lesquels un outil informatique est présent. La photographie, le sport, les jeux vidéo et la communication sont des exemples parmi d'autres. Le data mining est un domaine de l'informatique utilisé dans de nombreux contextes par exemple. On l'utilise pour l'étude du comportement humain ou animal ou afin de prédire la météorologie.

Dans le cadre d'un remaniement de l'enseignement de l'informatique dans les collèges de Genève, des cours ayant pour but d'introduire l'informatique sans utiliser les mathématiques ont été aménagés. Pour ce faire, de nouveaux outils pédagogiques doivent être développés. L'outil FunBlocks [7] est un outil développé dans le but de sensibiliser l'élève à la programmation informatique. Il consiste à manipuler des termes et des expressions, sous forme de bloc, afin de représenter des données et des instructions. Funblock utilise la programmation fonctionnelle contrairement à d'autres programmes visant à offrir un environnement visuel de programmation qui est basé sur la programmation impérative. La programmation fonctionnelle qui consiste à représenter le programme sous la forme de fonctions composées d'expressions plus complexes. Le but est ainsi de mieux pouvoir définir et manipuler des structures de données. Cette représentation se rapproche beaucoup de la manière dont l'algèbre est enseignée au collège de Genève.

FunBlock permet d'avoir un moyen intuitif d'écrire un programme afin de représenter un état et ses dérivations. Les blocs sont une représentation simple et claire, mais pas forcément familière pour un jeune élève. Que représente un bloc dans un autre bloc? Pourquoi deux blocs sont équivalents? Il a donc été suggéré d'introduire une image qui donnerait une interprétation des blocs.

Ainsi, l'utilisateur aura un outil plus familier pour comprendre les mécanismes que FunBlocks présente. L'extension FunView a pour objectif de rajouter une représentation graphique à l'état présent dans FunBlocks. Il a donc été nécessaire de trouver une représentation simple d'un bloc, mais gardant des structures très présentes en informatique comme une liste.

Ces objectifs en tête, il a donc fallu comprendre le fonctionnement de FunBlock pour construire FunView et définir son domaine. Une grammaire permettant de représenter l'état sous une forme graphique a été créée. Cette grammaire définit l'ensemble des représentations graphiques réalisables à l'aide de FunView. Funview pourra traduire un bloc accepté par cette grammaire en une série d'instructions. Ces instructions seront envoyées à un canvas qui créera la représentation graphique.

## 1.2 État de l'art

Comme dit précédemment, l'apprentissage de la programmation présente une grande difficulté de compréhension initiale. Des lignes de code ne sont pas très attrayantes avec la syntaxe qu'imposent la plupart des langages de programmation actuelles. Ainsi, réduire le code à de simples blocs sans aucune syntaxe verbeuse simplifierait l'introduction à la programmation pour un néophyte. Plusieurs études ont été réalisées dans ce domaine et viennent appuyer cette prédiction [9] [6]. Il est quand même à noter que cette solution n'est pas une panacée et perd en efficacité plus l'apprentissage s'approfondit.

Actuellement, il existe des outils pédagogiques proposant une expérience d'apprentissage de l'informatique de manière engageante à l'aide de blocs et d'image. Parmi les plus connus, nous avons MakeCode [5], Blockly [2], Scratch [3].

MakeCode est un programme open source offrant la possibilité d'apprendre à coder à l'aide de jeux, de programme simple et un résultat direct. L'apprenant peut manipuler des blocs représentant des lignes de code ou directement écrire dans un éditeur de texte à l'instar de FunBlock. Le résultat d'un programme sera représenté dans une grille de LED. MakeCode est principalement orienté vers l'apprentissage du langage Javascript [4].

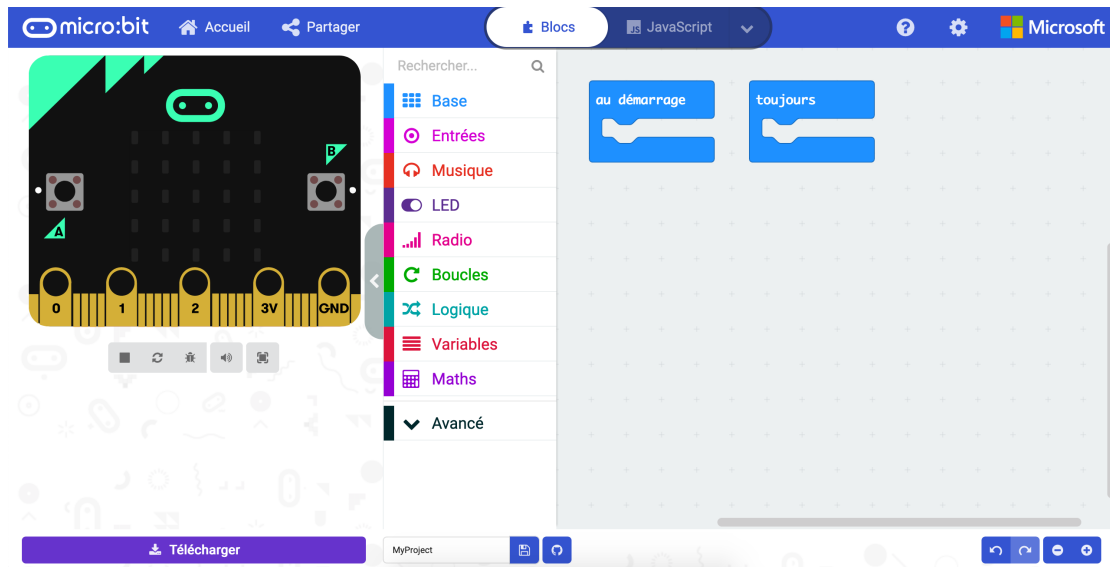


Figure 1.1: L'interface de MakeCode  
Source: Éditeur de MakeCode

Blockly est une librairie permettant d'ajouter un éditeur visuel à un site ou une application. IL se base sur des blocs pouvant s'imbriquer entre eux afin de représenter des programmes informatiques. Blockly peut transformer ses bloc en plusieurs langages de programmation : JavaScript, Python, PHP, Lua et Dart entre autres.

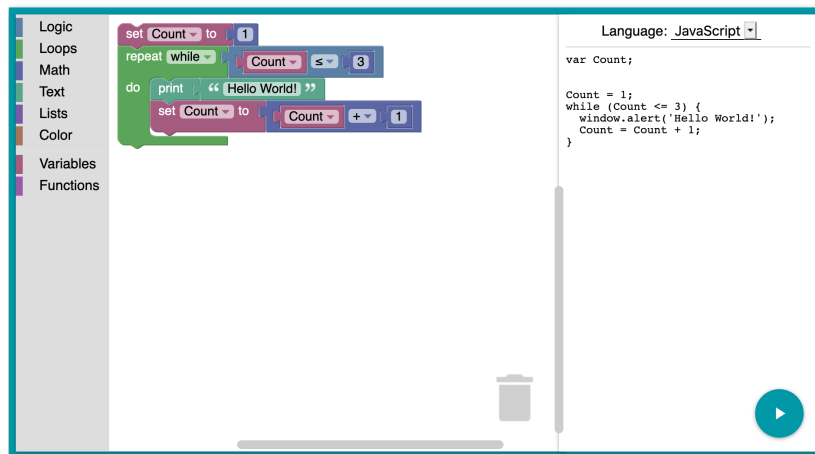


Figure 1.2: L'interface de Blockly  
Source: Site developers.google.com

Scratch est un programme open source permettant de créer des images, des animations, des histoires, des jeux et de la musique. Il se base sur le principe de bloc afin de produire des interactions avec des images ou des sons. Le but est d'introduire les jeunes enfants à des concepts informatiques et mathématiques.

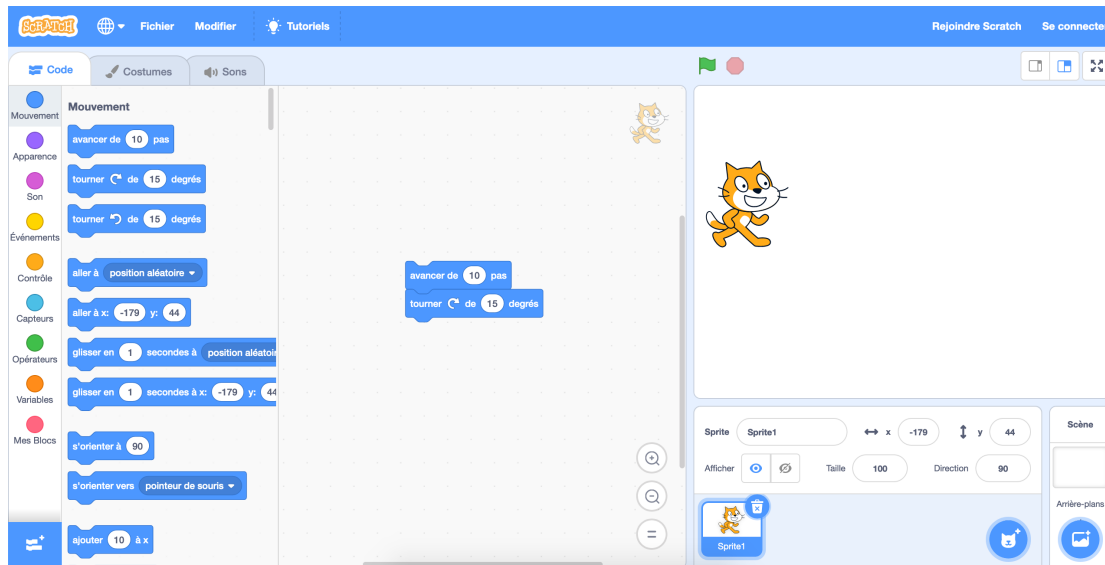


Figure 1.3: L'interface de Scratch  
Source: Tutoriel de Scratch

Ces programmes sont très complets et ils en existent d'autres proposant des solutions plus ou moins différentes. FunBlocks n'a pas l'ambition de proposer un produit capable de rivaliser avec MakeCode et Scratch. Son atout est plutôt la programmation fonctionnelle sur laquelle il se base. En effet, MakeCode et Scratch se basent sur la programmation impérative qui utilise une séquence d'instruction. Les instructions doivent être exécutées dans l'ordre dans lequel elles sont présentées. Il existe également des programmes orientés vers la programmation fonctionnelle, mais ils ne sont pas tournés vers l'apprentissage de la programmation. Le but de FunBlocks en utilisant la programmation fonctionnelle est de proposer une approche similaire à la façon dont est enseignée l'algèbre à l'école. FunBlocks a été développé pour être utilisé dans le cadre de l'enseignement de l'informatique au collège. Ainsi, Funblocks proposera une solution alternative à d'autres environnements de programmations visuelles pour introduire un élève au domaine de l'informatique.

# Chapitre 2: FunBlocks

---

Au début du projet, Funblocks se présentait comme suit:

- Un onglet Éditer : cet onglet de départ permet de créer son état initial et d'introduire des règles de transition. Il possède 2 modes, Blocks (Figure 2.1) et Text (Figure 2.2). Les deux modes sont liés et un changement dans l'un modifie l'autre.
  - Blocks permet de manipuler directement les objets à partir de la liste présente sur la gauche. L'utilisateur peut glisser une expression, une variable ou une règle dans les zones prévues à droite afin de construire un état. Toutes expressions ou règles peuvent être enlevées en les glissant sur l'icône de la poubelle en bas à gauche. En cliquant sur un bloc dans une règle ou l'état initial, l'utilisateur peut taper au clavier le label de ce bloc. Des blocs peuvent être glissés dans un autre bloc afin de créer un sous-bloc.
  - Le mode Text est un éditeur de texte. Il permet de représenter les objets utilisés dans le mode Blocs sous forme de code. Le mot clef *init* déclare l'état initial et le mot *case* une règle. Les parenthèses définissent un sous bloc.

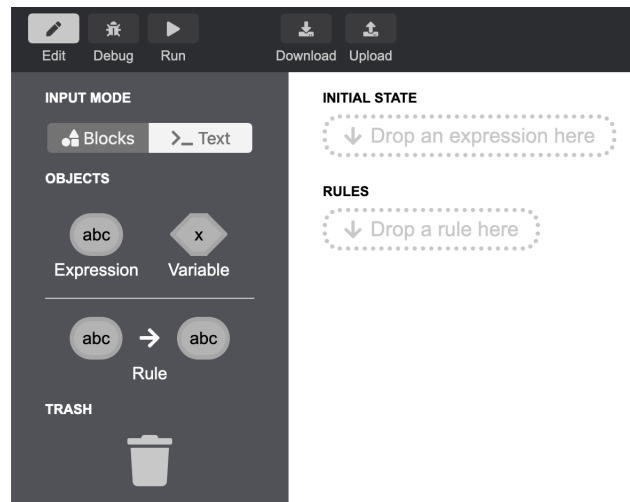


Figure 2.1: Onglet Edit dans le mode Blocks.

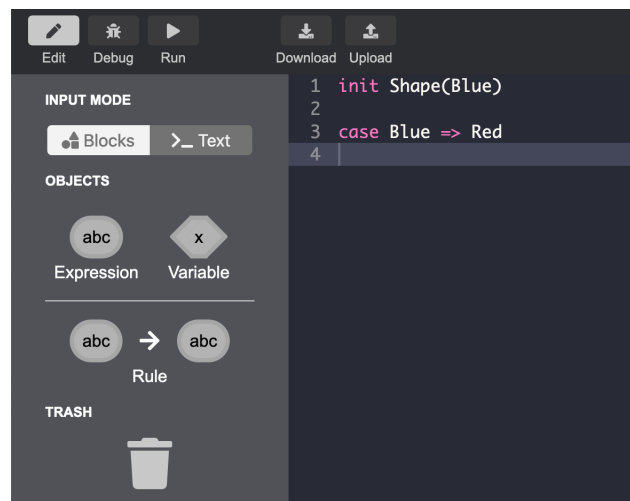


Figure 2.2: Onglet Edit dans le mode Text

- Un onglet Debug : Il permet de modifier l'état en fonction des règles établies dans le premier onglet. Pour appliquer une règle à un terme de notre état initial, il faut cliquer sur la règle puis sur le terme désiré. Si le terme correspond au membre de gauche de la règle, alors le terme est remplacé par la partie de droite de la règle. Un système d'historique permet de revenir à un état précédent (Figure 2.3). Un petit cercle est ajouté à la ligne d'historique à chaque application d'une règle.
- Un onglet Run : en cours de développement.



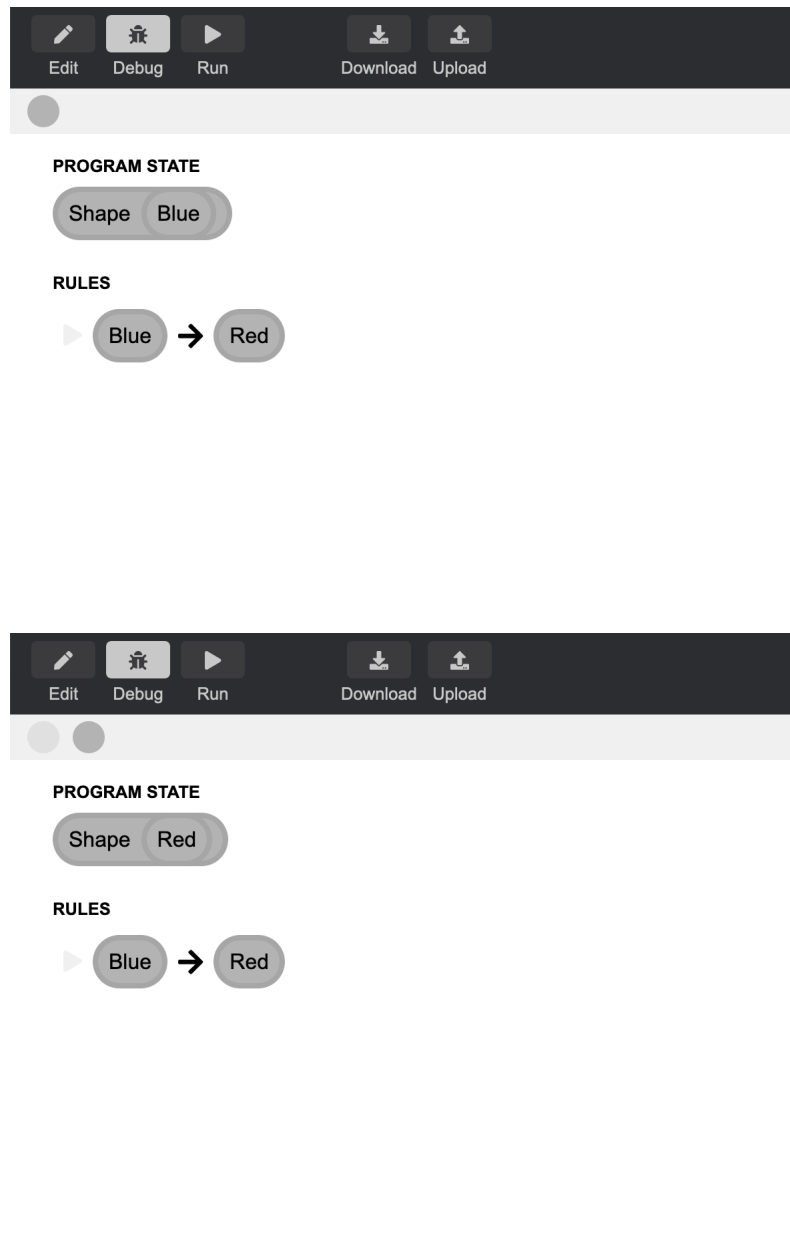


Figure 2.3: Onglet Debug avec l'état avant et après l'application d'une règle

L'état dans Funblock est représenté de trois manières différentes : sous forme de Blocs, sous forme de texte et de représentation graphique grâce à FunView. Le but premier de FunView est de représenter à tout moment l'état décrit dans Funblocks sous forme graphique. L'état est donc un élément qui doit être le même dans FunBlocks et dans FunView, car tout ce qui est exprimé dans cet état doit avoir un sens dans sa représentation graphique.

Dans Funblocks, l'état est présent sous la forme d'un arbre de syntaxe abstraite, ou AST pour abstract syntax tree. C'est un arbre dont les noeuds sont les opérateurs et les feuilles sont les opérandes. Ici, les noeuds sont des expressions et les feuilles sont des termes. Un terme est la représentation d'une expression. Une expression est une extension d'un terme. Tous les attributs d'un terme et d'une expression ne seront pas détaillés car une grande partie n'est pas utile au développement de FunView. Seuls les éléments utilisés dans FunView le seront.

Un terme possède un id et un label. L'id permet au terme d'être unique. Le label est le texte écrit dans un Bloc. Une expression possède les mêmes attributs qu'un terme. Cependant, une expression possède en plus un tableau de termes qui sont ses sous termes. Sur la figure 2.4, on peut voir la représentation *Shape* et ses sous termes ayant les labels *Square* et *Blue*.

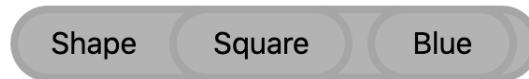


Figure 2.4: L'expression Shape avec ses deux sous-termes Square et Blue.

Lorsqu'une règle est appliquée à un état, ou plus précisément à un terme, l'état dans FunBlocks change. Ainsi, le changement aura un impact sur le label et les sous termes du terme. On peut voir un exemple grâce aux figures 2.5 et 2.6. On applique la règle modifiant un bloc Blue en bloc Red.

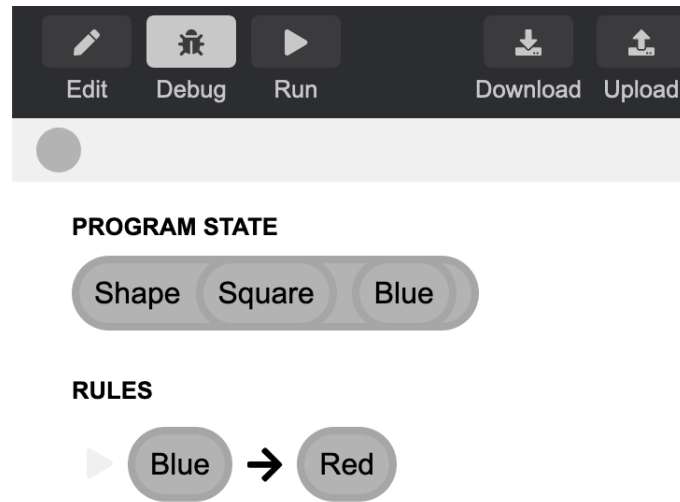


Figure 2.5: Notre programme avec une règle.

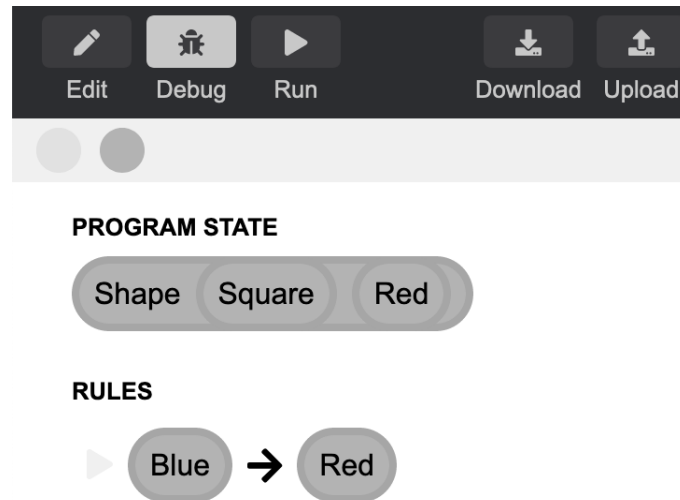


Figure 2.6: Ici, la règle a été appliquée à un sous-terme de notre programme.

# Chapitre 3: Le module FunView

---

Le principe de Funview est simple : un canvas HTML [1] sur lequel on dessine nos formes et un parseur qui permet de traduire notre état en un ensemble de formes. On passe donc de l'état sous forme AST à un canvas HTML.

Le parseur prend en entrée l'expression parent de l'état. Il en récupérera les enfants grâce à la structure d'une expression. Le parseur accepte un état respectant une certaine grammaire. Un fois cela fait il transmet la suite d'action à prendre à une classe qui modifiera le canvas en conséquence. La figure 3.1 montre le travail de FunView dans son ensemble.

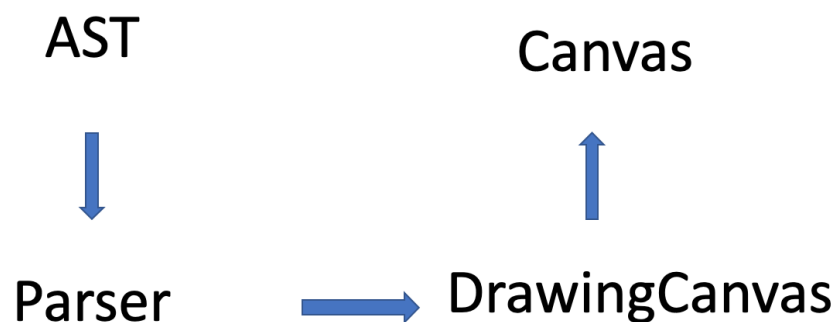


Figure 3.1: Le travail de Funview

## 3.1 La grammaire

L'objectif par l'ajout d'une représentation graphique, était de montrer l'état sous une autre forme. Il a fallu décider quelle forme allait prendre l'état. Le choix devait être simple, compréhensible par un élève et pouvant représenter un état avec ses caractéristiques. Il a été décidé de pouvoir afficher des formes géométriques et de pouvoir exprimer leurs positions par rapport à d'autres formes. Cette représentation permet de facilement représenter des concepts informatiques comme des listes. Une grammaire régissant les formes et les positions a dû être créée. L'idée est d'avoir

un langage qui permet de traduire des expressions en images. L'utilisateur aura à sa disposition des formes, des couleurs et des fonctions pour assigner une position à chaque forme.

La grammaire a connu plusieurs versions. Les débuts étaient très basiques et ont surtout permis de se familiariser avec les règles du canvas HTML.

```
<etat> -> <terme>
<terme> -> <fonction> ( <terme> , <terme> ) |
Shape ( <form>, <color>)
<fonction> -> Over | Under | LeftOf | RightOf
<form> -> Square | Circle | Triangle
<color> -> Red | Blue | Green | Yellow
```

Les fonctions permettent de placer une forme en fonction d'une autre. *Shape* permet de déclarer une forme d'une certaine couleur.

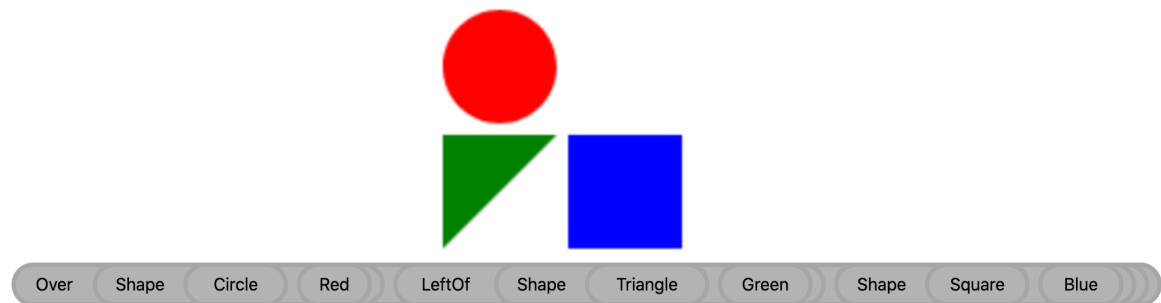


Figure 3.2: Un état accepté par la grammaire sous sa forme graphique et sa forme en block. Le cercle se retrouve au dessus, grâce au terme *Over*, de l'ensemble décrit dans le terme *LeftOf*.

La fonction *Cons* a ensuite été rajoutée pour avoir un système de liste. Elle est semblable à *LeftOf*. Une forme *empty* a aussi été ajoutée afin de définir la fin d'une liste.

Par la suite, deux nouvelles fonctions ont été rajoutées. Ces nouvelles fonctions sont des versions améliorées des précédentes et bien plus puissantes. Elles ont une signature différente des anciennes fonctions. La grammaire a donc été légèrement modifiée.

```
<etat> -> <fonction>
<fonction> -> <relation> ( <fonction> , <fonction> ) |
    Shape ( <form>, <color>) |
    Draw ( <fonction>, <fonction>, Offset(X,Y) ) |
    Group (Forms (<shapes>) , Pos( <pos>))
```

```

<shapes> -> <shapes> ,Shape( <form>, <color>) |
        Shape( <form>, <color>)
<pos> -> <pos> , X,Y | X,Y
<relation> -> Over | Under | LeftOf | RightOf
<form> -> Square | Circle | Triangle
<color> -> Red | Blue | Green | Yellow | Black

```

Si <color> est vide alors la couleur par défaut est noir.  
 <pos> doit avoir une taille deux fois plus  
 grande que <shapes> afin d'avoir un X et un Y par Shape.

Nous avons donc *Draw* (Figure 3.3) et *Group* (Figure 3.4) qui permettent respectivement de placer deux formes avec une certaine position de différence et de placer une liste de formes a des positions indépendantes entre elles.

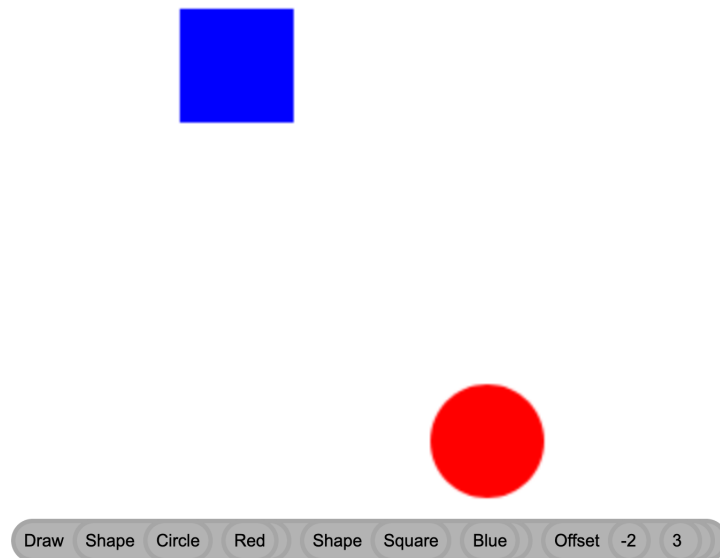


Figure 3.3: Exemple de la fonction Draw. Le carré se trouve à la position (-2, 3) par rapport au cercle

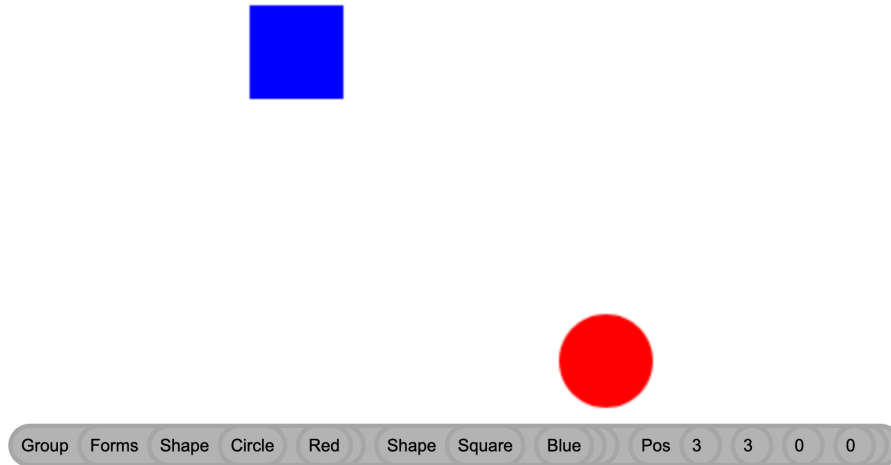


Figure 3.4: Exemple de la fonction Group. La position du cercle est (3, 3) et celle du carré est (0,0).

## 3.2 La sémantique

Une forme simple est une paire dont le premier élément décrit sa forme géométrique et le second élément décrit sa couleur. Une forme est contenue dans un carré de dimension de 55 pixels sur 55 pixels. Une forme a toujours une envergure de 50 pixels. C'est pourquoi nous avons ajouté 5 pixels de marge entre chaque forme. On définit  $G = \{Square, Circle, Triangle\}$  comme l'ensemble de formes géométriques supportées par FunView. Un exemple de ses formes se trouve sur la figure 3.2. Les couleurs supportées par FunView sont définies par  $C = \{Green, Red, Yellow, Blue, Black\}$ . Une forme simple  $s$  est définie comme un élément de  $G \times C$ . Ainsi une forme  $s = (Square, Green)$ , donnera un carré vert. C'est que l'on observe sur la figure 3.5

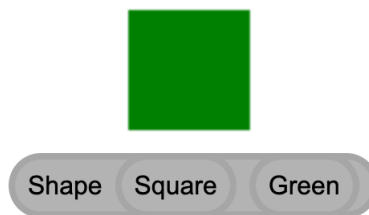


Figure 3.5: Le terme Shape (Square Green) représente  $s = (Square, Green)$  et produit donc un carré vert sur le canvas.

Viens ensuite les formes composées. Elles sont produites à l'aide des opérateurs  $O = \{Draw(s_1, s_2, Offset(x, y)), Group(Forms(n), Pos(l)), Over(s_1, s_2),$

$Under(s_1, s_2), LeftOf(s_1, s_2), RightOf(s_1, s_2)\}$  où  $s_1, s_2 \in S$ ,  $n$  est une liste d'éléments appartenant à  $S$  et  $l$  est une liste d'entier d'une taille paire, et des formes simples. Ainsi les formes composées sont définies comme des éléments de l'ensemble  $O \times G \times C$ . L'ensemble de toutes les formes est l'ensemble  $S = G \times C \times O$ .

La position d'une forme est défini par ses coordonnées ou sa position par rapport à une autre forme. L'unité des coordonnées est d'une longueur de 55 pixels afin de pouvoir contenir une forme par unité. Les axes ne sont pas affichés car il n'a pas été jugé important de le faire. Cependant en dehors de l'utilisation de la fonction *Group*, l'origine n'a pas d'importance car la position est déclarée relativement à d'autres formes uniquement.

La manière dont sont interprétés les formes et les fonctions est régit par la sémantique suivante :

- $g \in G \Rightarrow draw(g) \in I$
- $c \in C \Rightarrow paint(c) \in I$
- $i \in \mathbb{Z} \Rightarrow X(i) \in I$
- $i \in \mathbb{Z} \Rightarrow Y(i) \in I$

La liste d'instructions  $i = \{draw(Circle), paint(Red), Y(), draw(Triangle), paint(Green), Y(-1), X(), draw(Square), paint(Blue), X(-1)\}$  ordonne au canvas de dessiner la figure 3.6.



Figure 3.6: D'après la liste  $i$  le canvas, dessine un cercle, le peint en rouge, déplace son pointeur d'un cran vers le bas, dessine un triangle, le peint en vert, remonte le pointeur d'un cran, déplace le pointeur d'un cran vers la droite, dessine un carré, le peint en bleu et remet le pointeur à sa position d'origine. Ce résultat peut être produit à partir de la forme composée  $s = LeftOf(Over(Shape(Circle, Red), Shape(Triangle, Green)), Shape(Square, Blue))$



Ces instructions correspondent à l'API d'un canvas HTML.  $draw(g)$  dessine la forme  $g$ .  $paint(c)$  peint une forme.  $X(i)$  et  $Y(i)$  déplacent le pointeur du canvas.

Ainsi, nous avons une liste  $N$  pouvant contenir des éléments appartenant à  $S$  et à  $I$ .  $N$  sera modifié en suivant les règles d'inférences décrites plus bas. À terme, la liste  $N$  sera vidée et une liste d'instructions finale  $F = (I)^*$  sera envoyé au canvas HTML. On a donc :

- $N = (S \cup I)^*$ . La liste de départ
- $F = (I)^*$ . La liste d'instruction finale pour le canvas HTML.

La liste  $N$  est une liste composée de formes simples ou composées appartenant à  $S$  et d'instructions  $I$ . En joignant les deux, on permet d'évaluer une fonction tout en gardant l'ordre relatif aux instructions déjà définies. Au départ, contient la forme composée que l'on cherche à traduire en instruction de canvas HTML et la liste  $F$  est vide. On évalue le premier terme de la liste  $N$  et on applique la règle d'inférence. Suivant la règle, une instruction est rajoutée à  $N$  afin de garder l'ordre d'évaluation par rapports autres termes. On recommence jusqu'à ce que  $N$  soit vide. Quand c'est le cas, la liste  $F$  contient toutes les instructions nécessaires aux canvas HTML pour dessiner l'image.

Sémantique d'évaluation :

On définit :

$n, m \in N$ ,  $c \in C$ ,  $g \in G$ ,  $f \in F$ ,  $x, y \in \mathbb{Z}$ ,  $s_i \in S$ ,  $l$  est une liste d'entier d'une taille paire. Le symbole  $\cdot$  représente la concaténation.

Règles d'inférences :

Regardons d'abord les règles où  $s_0$  ajoute une instruction à  $F$  :

$$\frac{s_0 = X()}{< s_0 \cdot n, f > \Rightarrow < n, f \cdot X() >} X()$$

$$\frac{s_0 = X(x)}{< s_0 \cdot n, f > \Rightarrow < n, f \cdot X(x) >} X(x)$$

Les règles d'inférences pour  $Y()$  et  $Y(y)$  sont semblable à celle de  $X()$  et  $X(x)$

$$\frac{s_0 = c}{< s_0 \cdot n, f > \Rightarrow < n, f \cdot \text{paint}(c) >} \text{paint}$$

$$\frac{s_0 = g}{< s_0 \cdot n, f > \Rightarrow < n, f \cdot \text{draw}(g) >} \text{draw}$$

La fonction Shape :

$$\frac{t_0 = \text{Shape}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_1 \cdot s_2 \cdot n, f >} \text{Shape}$$

Les opérateurs :

$$\frac{s_0 = \text{LeftOf}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_1 \cdot X() \cdot s_2 \cdot X(-1) \cdot n, f >} \text{LeftOf}$$

$$\frac{s_0 = \text{Shape}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_2 \cdot X() \cdot s_1 \cdot X(-1) \cdot n, f >} \text{RightOf}$$

$$\frac{s_0 = \text{Over}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_1 \cdot Y() \cdot s_2 \cdot Y(-1) \cdot n, f >} \text{Over}$$

$$\frac{s_0 = \text{Under}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_2 \cdot Y() \cdot s_1 \cdot Y(-1) \cdot n, f >} \text{Under}$$

$$\frac{s_0 = \text{Cons}(s_1, s_2)}{< s_0 \cdot n, f > \Rightarrow < s_1 \cdot X() \cdot s_2 \cdot X(-1) \cdot n, f >} \text{Cons}$$

$$\begin{array}{c}
\frac{s_0 = \text{Draw}(s_1, s_2, \text{Offset}(x, y)), x > 0, y > 0}{< s_0 \cdot n, f > \Rightarrow < s_1 \cdot \text{Offset}(x, y) \cdot s_2 \cdot n, f >} \text{Draw} \\
\\
\frac{s_0 = \text{Draw}(s_1, s_2, \text{Offset}(x, y)), x < 0 \vee y < 0}{< s_0 \cdot n, f > \Rightarrow < s_2 \cdot \text{Offset}(x, y) \cdot s_1 \cdot n, f >} \text{DrawNeg} \\
\\
\frac{s_0 = \text{Offset}(|x|, |y|)}{< s_0 \cdot n, f > \Rightarrow < n, f \cdot X(x) \cdot Y(y) >} \text{Offset} \\
\\
\frac{s_0 = \text{Group}(\text{Forms}(s_1 m), \text{Pos}(x, y, l))}{< s_0 \cdot n, f > \Rightarrow < X(x) \cdot Y(y) \cdot t_1 \cdot X(-x) \cdot Y(-y) \cdot \text{Group}(\text{Forms}(m), \text{Pos}(l)) \cdot n, f >} \text{Group} \\
\\
\frac{s_0 = \text{Group}(\text{Forms}(\varepsilon), \text{Pos}(\varepsilon))}{< s_0 \cdot n, f > \Rightarrow < n, f >} \text{Group}
\end{array}$$

La règle terminal :

$$\frac{}{< \varepsilon, f >} \text{end}$$

La règle *Group* fonctionne de manière récursive afin de parcourir toute la liste contenue dans *Forms*. Il est à noter que cette règle ainsi que les règles *DrawNeg*, *Draw*, *Cons*, *Under*, *Over*, *RightOf*, *LeftOf* et *Shape* n'ont aucun impact direct sur la liste *I*. Ce sont des règles servant à établir l'ordre des instructions.

Par exemple, l'évaluation de l'état de la figure 3.3 se fera ainsi :

$$\begin{array}{c}
\frac{}{< \varepsilon, \varepsilon > \Rightarrow < \varepsilon, \varepsilon >} \text{end} \\
\frac{}{< \text{Red}, \varepsilon > \Rightarrow < \varepsilon, \text{paint}(\text{Red}) >} \text{paint} \\
\frac{}{< \text{Circle} \cdot \text{Red}, \varepsilon > \Rightarrow < \varepsilon, \text{draw}(\text{Circle}) \cdot \text{paint}(\text{Red}) >} \text{draw} \\
\frac{}{< \text{Shape}(\text{Circle}, \text{Red}), \varepsilon > \Rightarrow < \varepsilon, \text{draw}(\text{Circle}) \cdot \text{paint}(\text{Red}) >} \text{Shape} \\
\frac{}{< \text{Offset}(-2, 3) \cdot \text{Shape}(\text{Circle}, \text{Red}), \varepsilon > \Rightarrow < \varepsilon, X(-2) \cdot Y(3) \cdot \text{draw}(\text{Circle}) \cdot \dots >} \text{Offset} \\
\frac{}{< \text{Blue} \cdot \text{Offset}(-2, 3) \cdot \text{Shape}(\text{Circle}, \text{Red}), \varepsilon > \Rightarrow < \varepsilon, \text{paint}(\text{Blue}) \cdot X(-2) \cdot Y(3) \cdot \dots >} \text{paint} \\
\frac{}{< \text{Square} \cdot \text{Blue} \cdot \text{Offset}(-2, 3) \cdot \text{Shape}(\text{Circle}, \text{Red}), \varepsilon > \Rightarrow < \varepsilon, \text{draw}(\text{Square}) \cdot \text{paint}(\text{Blue}) \cdot \dots >} \text{draw} \\
\frac{}{< \text{Shape}(\text{Square}, \text{Blue}) \cdot \text{Offset}(-2, 3) \cdot \text{Shape}(\text{Circle}, \text{Red}), \varepsilon > \Rightarrow < \varepsilon, \text{draw}(\text{Square}) \cdot \text{paint}(\text{Blue}) \cdot \dots >} \text{Shape} \\
\frac{}{< \text{Draw}(\text{Shape}(\text{Circle}, \text{Red}), \text{Shape}(\text{Square}, \text{Blue}), \text{Offset}(-2, 3)), \varepsilon > \Rightarrow < \varepsilon, \text{draw}(\text{Square}) \cdot \text{paint}(\text{Blue}) \cdot \dots >} \text{DrawNeg}
\end{array}$$

La construction de la règle se fait en partant du membre gauche en bas. On remonte en appliquant nos règles d'inférence. On remarque que certaines règles n'ont aucun impact sur le membre de droite. C'est le cas pour *DrawNeg* et *Shape*. Il en est de même pour toutes les règles énoncées précédemment n'ayant pas d'impact direct sur la liste *I*. Le membre de gauche change lorsqu'une évaluation est complétée. Par exemple, on rajoute l'instruction *draw(Circle)* à la liste *I* lorsque la règle *draw* est validée. Le résultat final est le membre en bas à droite où *N* est vide et *I* contient alors les instructions qui seront traitées par la classe *DrawingCanvas* pour dessiner le système de forme voulue.

### 3.3 Le parseur

Le parseur applique donc nos règles de grammaire à l'état sous forme AST. Pour cela, il utilise une méthode récursive qui traduit chaque étage par une action. La première chose à savoir c'est que le canvas HTML fonctionne selon un certain ordre. Par exemple, si l'on veut colorier une forme il faut d'abord donner la forme à dessiner puis donner la couleur pour ensuite lui dire de remplir la forme. Si l'on veut déclarer la position d'une forme, il faut donner la position en premier. Ainsi, le parseur aura comme rôle principal d'établir l'ordre dans lequel les instructions sont envoyées au canvas.

La méthode récursive est appelée *explore* et est une méthode de la classe Drawn-State. Cette classe contient uniquement ladite méthode et un appel à la classe manipulant le canvas, DrawingCanvas. La manière de procéder d'*explore* est d'agir en fonction du label de l'expression courante (le noeud courant de l'AST). D'après le label, elle demandera à DrawingCanvas d'agir en conséquence ou bien elle se appellera elle-même selon l'ordre adéquat.

Sur la Figure 3.7, On peut voir la transformation que permet FunView.

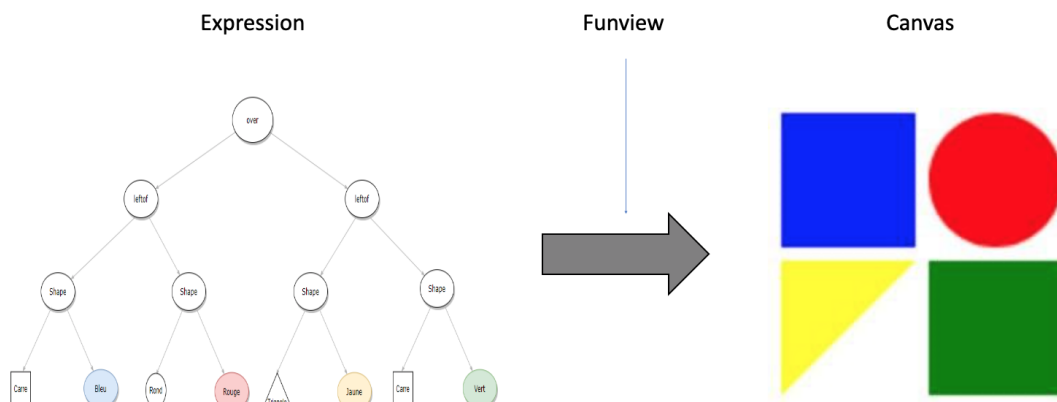


Figure 3.7: Exemple de l'état sous forme AST et sa représentation graphique

La méthode *explore* parcourt l'arbre en profondeur. Dans la figure 3.7, elle commence à la racine *Over* et informe par conséquent la classe DrawingCanvas que le premier enfant de *Over* sera au-dessus du deuxième. Elle continue son exploration sur le premier enfant. En rencontrant l'expression *LeftOf*, elle informe DrawingCanvas que le premier enfant de *LeftOf* se trouvera à gauche du deuxième enfant. Elle arrive sur la fonction *Shape* la plus à droite de la figure 3.7. *Shape* annonce simplement que ses enfants sont des feuilles. Une fois arrivée à la feuille *Square* la plus à gauche, *explore* demande à la classe DrawingCanvas de

dessiner un carré. Elle passe ensuite au prochain noeud à explorer, le noeud bleu, la fonction *Shape* ne donnant pas d'instructions supplémentaires. Ainsi, *DrawingCanvas* saura que le carré précédemment commandé est de couleur bleue. La fonction *explore* remonte ensuite sur l'expression *LeftOf* et visite son deuxième enfant. L'exploration continue pendant que *DrawingCanvas* modifie le canvas. Une fois les enfants du premier *LeftOf* parcourus, *explore* passe au deuxième enfant de *Over*.

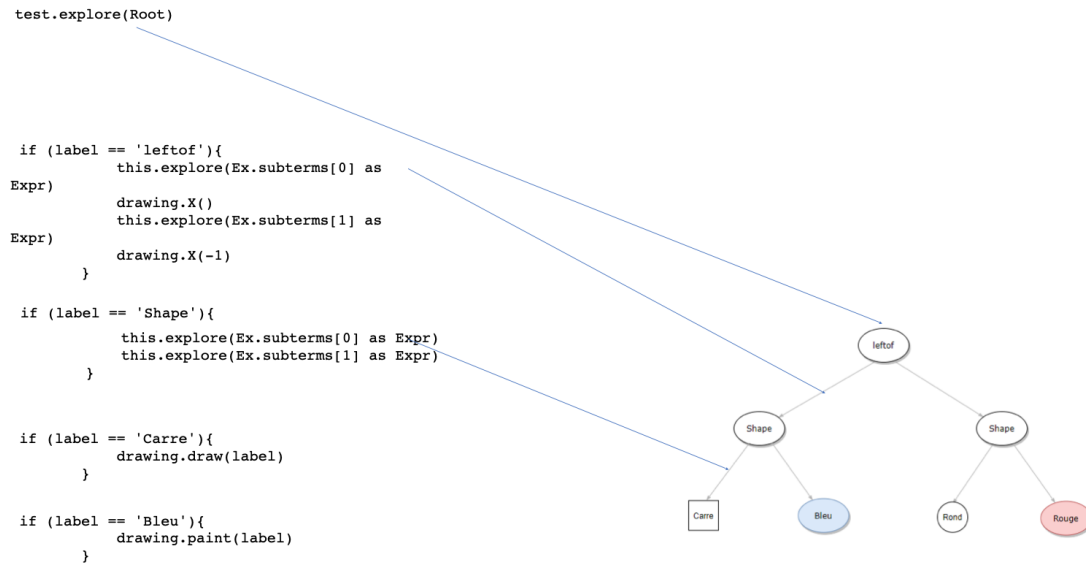


Figure 3.8: Exemple de parcours par la méthode recursive explore

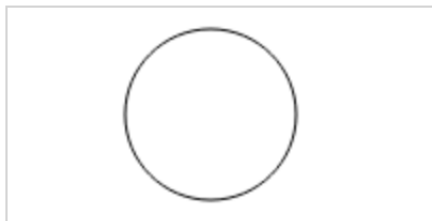
La figure 3.8 illustre comment *explore* se rappelle elle-même avec une nouvelle expression afin d'explorer l'arbre. On voit que la fonction *LeftOf* ordonnera à *DrawingCanvas* placer le deuxième sous terme à droite du premier à l'aide de la méthode *X()*.

### 3.4 Le canvas HTML

Le canvas HTML est un objet HTML permettant de modifier une zone graphique à l'aide d'un script. Il est souvent utilisé pour modifier une image ou bien jouer des animations. Au début du projet, il avait été question d'utiliser le format SVG [10] afin d'avoir des images vectorielles. Cependant, il a été décidé d'utiliser le canvas HTML car il demanderait moins de travail de recherche de par son fonctionnement très simple.

Le canvas nécessite une mise en place avant de pouvoir travailler dessus. En premier temps, un objet canvas est ajouté au fichier *index.html*. Puis il est modifié

et manipulé dans la classe `DrawingCanvas` qui se charge de toute la partie dessin. Le canvas fonctionne à partir d'un pointeur qui servira de pinceau afin de tracer une forme. La position de ce pointeur est déterminé à partir de ces coordonnées sur le canvas. Le canvas est une zone sur laquelle on peut dessiner à l'aide du pointeur. L'origine est le coin supérieur gauche du canvas. La valeur de X grandit en allant vers la droite et la valeur de Y augmente en descendant. En déplaçant le pointeur ou en lui donnant des formes à effectuer, on peut créer un dessin. Il est également possible de rajouter des couleurs, qu'elles soient unies ou dégradées. Le site de [W3schools.com](http://W3schools.com) sur les canvas HTML présente toutes les possibilités qu'offre cet objet. Dans `FunView`, nous utiliserons les méthodes `arc`, `rect` et `lineTo` pour dessiner respectivement un cercle, un rectangle et un triangle. Il est également possible de dessiner du texte à l'aide de la méthode `fillText`.



## Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(95, 50, 40, 0, 2 * Math.PI);
ctx.stroke();
</script>
```

Figure 3.9: Exemple du script complet produisant un cercle et son résultat. Les deux premiers arguments d' `arc` sont les coordonnées du centre du cercle. La troisième est la taille du rayon et les deux dernières sont les valeurs d'angle de départ et d'arrivée.

Source : [W3schools.com](http://W3schools.com)

La partie la plus compliquée était le positionnement des formes. Il fallait déplacer le pointeur désignant le début du dessin à gauche, à droite, en haut et en bas et pouvoir retourner à la position précédente après avoir dessiné la forme. Les méthodes *X()* et *Y()* de *DrawingCanvas* ont été désigné dans ce but. Elles permettent de se déplacer d'une distance égale à l'envergure d'une forme, soit 55 pixels. Dans ces méthodes, un moyen de modifier la taille du canvas a été rajouté si l'ensemble de formes sortait des limites.

La classe *DrawingCanvas* contient également les méthodes *draw* et *paint* qui sont appelées par la classe *Parser*. La méthode *draw* permet de construire la figure en fonction de la chaîne de caractères donnée en argument. Elle utilise pour cela les méthodes *arc*, *rect* et *lineTo*. *draw* permet également d'afficher si une erreur a été commise lors de l'écriture des blocs. Elle écrit un message d'erreur sur le canvas à l'aide de *fillText*. La méthode *paint* quant à elle assigne une couleur à la forme décrite précédemment en utilisant la méthode *fillStyle* du canvas *HTML*. Elle prend également une chaîne de caractères en argument afin de déterminer quelle couleur.

# Chapitre 4: Relier FunBlocks et FunView

---

## 4.1 Ajouter FunView au projet FunBlocks

Le projet Funview est un module du projet FunBlock. Ainsi, dans chaque fichier faisant appel à FunView, il a fallu importer le module *FUNVIEW*.

Il a fallu également rajouter le canvas HTML à l'ensemble FunBlocks. L'élément HTML canvas a donc été rajouté dans le fichier *index.html*. La figure 4.1 montre le résultat de l'ajout du canvas.

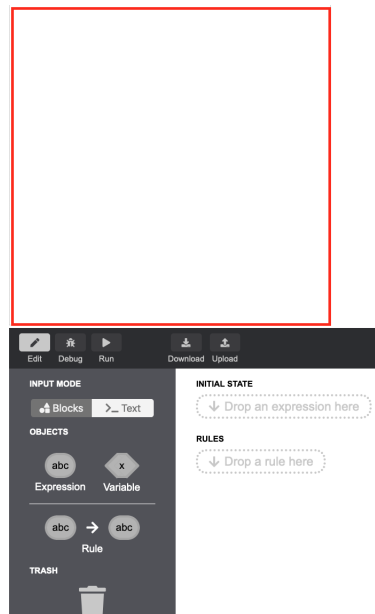


Figure 4.1: On voit ici la zone marquée en rouge réservée au canvas HTML au dessus de l'interface FunBlocks. C'est ici que sera dessiné les formes représentant l'état du programme.



Cet arrangement simpliste a été choisi afin d'éviter de perdre du temps au détriment de la fonctionnalité de l'ensemble.

Durant le développement isolé de FunView, la structure d'un état défini dans le module *AST* de FunBlock a été simulé en créant une version plus simple des expressions. Lors de l'ajout du module *FUNVIEW*, le module *AST* a été importé dans *FUNVIEW*.

Dès lors, FunView fonctionne dans FunBlocks, mais il faut qu'il soit réactif aux changements faits sur l'état dans FunBlocks.

## 4.2 Les appels à FunView

FunBlocks utilise la bibliothèque Javascript React [8] qui permet de créer une application web sur une seule page. La page réagit aux actions performedes sur ladite page. Une action peut être un clic ou un glisser-déposer. Ces actions produisent ce qu'on appelle un changement d'état. Ainsi, on peut naviguer entre plusieurs *View* grace à certains changements d'état. Dans FunBlocks, ces *View* sont appelée *Workspaces*.

La figure 4.2 montre les différentes *View*. Comme dit précédemment, on passe d'une *View* à l'autre à l'aide d'action. Ici, l'action est de cliquer sur le bouton Debug pour aller dans le *Workspace* Debug par exemple.

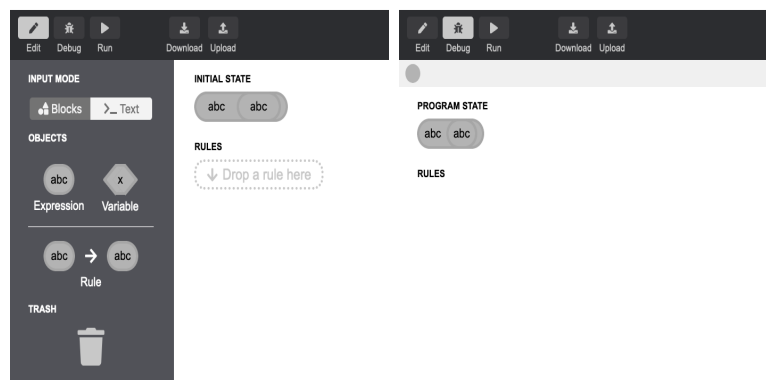


Figure 4.2: Le *Workspace* Edit et le *Workspace* Debug

Ainsi, déterminer quand FunBlock doit appeler FunView se résume à trouver quels changements d'état doivent avoir un impact sur FunView. Ces choix ont été fait de manière arbitraire et peuvent être sujets à de futurs changements. Il a donc été décidé d'appeler FunView lorsque :

- Le *Workspace* Debug est affiché;

- Une règle est appliquée à l'état du programme;
- On change l'état du programme en naviguant dans l'historique.

Les fonctions liées à ces trois actions se trouvent dans le dossier Reducers du module *UI*. La première, liée au *Workspace* Debug, se trouve dans le fichier *index.ts*, lorsque le changement vers le *Workspace* Debug est traité. L'application d'une règle et la navigation dans l'historique se trouvent dans le fichier *DebugContext.ts*. L'un se trouve dans le cas où un état est poussé et l'autre se trouve dans le cas du changement d'index d'historique.

Dans les trois cas, on instancie les classes *DrawingCanvas* et *Parser*. On vide le canvas car on veut éviter de superposer plusieurs formes. Enfin, on appelle la méthode *explore* de la classe *Parser* avec l'état du programme lors de l'action. Ainsi, *FunView* est opérationnel en collaboration avec *FunBlocks*.

# Chapitre 5: Conclusion

---

Ce projet avait pour but de proposer un service permettant d’avoir une représentation graphique d’un état au sens sémantique du terme. Cela afin de faciliter la compréhension et la manipulation de FunBlocks et ainsi produire un outil pédagogique pour l’enseignement de l’informatique au Collège de Genève.

Il a fallu pour cela créer une grammaire permettant un parallèle entre un état possiblement abstrait et un arrangement de forme. Cette grammaire possède plusieurs fonctions, bien qu’une seule soit essentielle et peut générer toutes les autres. Il a fallu un temps d’adaptation avant de pouvoir être à l’aise dans l’environnement de FunBlocks. C’est pourquoi les fonctions `LeftOf`, `RightOf`, `over` et `under` ont été développées au début. Elles étaient très basiques et sont, à la fin du projet, obsolètes. Elles ont néanmoins servi de terrain d’entraînement. Le rythme a assez rapidement augmenté une fois une certaine expérience acquise et le langage n’a plus posé de gros problèmes de compréhension.

La grammaire est un apport qui peut être amélioré. Avec ce système de forme et de positionnement, nous avons voulu proposer une représentation compréhensible de l’état. Cependant, ce n’est qu’une réponse parmi pleins d’autres. Un autre parallèle peut être ajouté entre un état et sa représentation graphique. D’autres idées pourraient émerger et devenir plus pertinentes.

Afin de comprendre cette grammaire, FunView a besoin d’un parser. Ce parser a été construit en une seule méthode *explore*. Actuellement, *explore* fonctionne en s’appelant elle-même et en lisant le label du noeud courant. Elle est donc très longue et redondante. Dans un souci de clarté, une série de plus petites méthodes, chacune régissant une fonction de la grammaire, serait envisageable.

Ce parser communique au travers d’une classe les instructions pour le canvas. Le canvas est simplement défini au-dessus de l’interface FunBlock. Une meilleure intégration du canvas serait un grand ajout. Une solution alternative au canvas serait également concevable. Il avait été fait mention du format SVG, mais

l'utilisation de fichiers d'image peut être également prometteuse.

Dans l'ensemble, le projet a eu un rythme assez régulier. La partie concernant la création de la grammaire a pris du temps, mais il était nécessaire de bien poser les bases de celle-ci pour la suite du projet. FunView a été fonctionnel assez rapidement ce qui a permis de revenir sur des détails de la grammaire et d'offrir plus de fonctionnalité. C'est ainsi que les fonctions *Draw* et *Group* ont vu le jour. La présentation du canvas ainsi que son incorporation ont été réalisés rapidement afin de pouvoir tester la fonctionnalité de l'ensemble.

Actuellement, Funblock complété par le module FunView propose une solution alternative à d'autres programmes tel que MakeCode et Scratch. Il reste cependant bien plus simple et à un stade où beaucoup d'éléments peuvent être rajoutés. Durant ce projet il n'a pas été question d'animation, ce que proposent MakeCode et Scratch. De plus, les créations graphiques possibles sur ces deux programmes sont poussés bien plus loin que celles de Funview. Ces points seraient des options à explorer. Un exemple de fonctionnalité à rajouter en plus serait de permettre une modification de la représentation graphique. Cela aurait un impact sur l'état représenté sous forme de bloc dans FunBlocks. Cette idée ne serait pas forcément compatible avec un canvas HTML, mais un développement de ce côté serait envisageable. En conclusion, Funview propose un début prometteur, mais nécessite un approfondissement de ses fonctionnalités afin de pouvoir proposer un outil pédagogique complet au service du Collège de Genève.

- 
- [1] *Canvas HTML*. URL = [https://fr.wikipedia.org/wiki/Canvas\(\*HTML\*\)](https://fr.wikipedia.org/wiki/Canvas_(HTML)), consulté en septembre 2020.
  - [2] Neil fraser. *Blockly*. URL = <https://developers.google.com/blockly/>, consulté en septembre 2020. 2013.
  - [3] MIT. *Scratch*. URL = <https://scratch.mit.edu/>, consulté en septembre 2020. 2006.
  - [4] *Javascript*. URL = <https://www.javascript.com/>, consulté en septembre 2020.
  - [5] Microsoft. *MakeCode*. URL = <https://makecode.microbit.org/>, consulté en septembre 2020.
  - [6] Thomas W. Price and Tiffany Barnes. “ICER ’15: Proceedings of the Eleventh Annual International Conference on International Computing Education Research”. In: (2015), pp. 91–99.
  - [7] Dimitri Racordon. *FunBlocks*. Github = <https://github.com/kyouko-taiga/FunBlocks>, consulté en septembre 2020.
  - [8] *React*. URL = <https://fr.reactjs.org/>, consulté en septembre 2020.
  - [9] Fukazawa Y. Saito D. Washizaki H. “Comparison of text-based and visual-based program-ming input methods for first-time learners”. In: (2017). Source : <https://www.informingscience.org/Publications/3775>, pp. 209–226.
  - [10] *SVG*. URL = [https://fr.wikipedia.org/wiki/Scalable\\_Vector\\_Graphics](https://fr.wikipedia.org/wiki/Scalable_Vector_Graphics), consulté en septembre 2020.