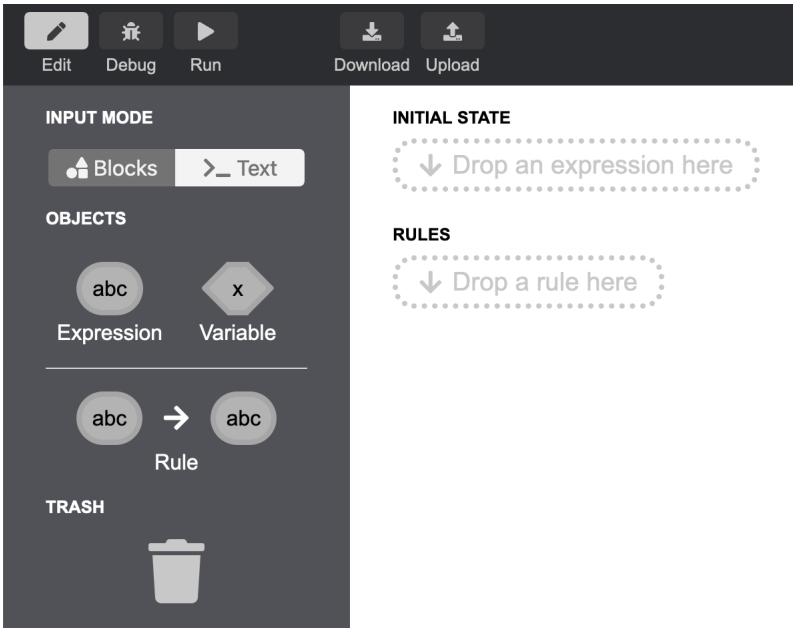
UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES

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

Chapitre 1: Introduction

1.1 Contexte

A 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 utilisé pour résoudre des problèmes sans calcul. Il suffit de voir tout 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 [4] 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 block, afin de représenter des données et des instructions. Funblock utilise la programmation fonctionnel contrairement à d'autre programme visant à offrir un environnement visuel de programmation qui sont 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 enseigné au collège de Genève.

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

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 block, 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 défini l'ensemble des représentations graphiques réalisables à l'aide de FunView. Funview pourra traduire un block accepté par cette grammaire en une série d'instruction. Ces instructions seront envoyées à un canvas qui créera la représentation graphique.

1.2 État de l'art

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

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 blocks représentant des lignes de code ou directement écrire dans un éditeur de texte. Le résultat d'un programme sera représenté dans une grille de LED. MakeCode est principalement orienté vers l'apprentissage du langage Javascript.

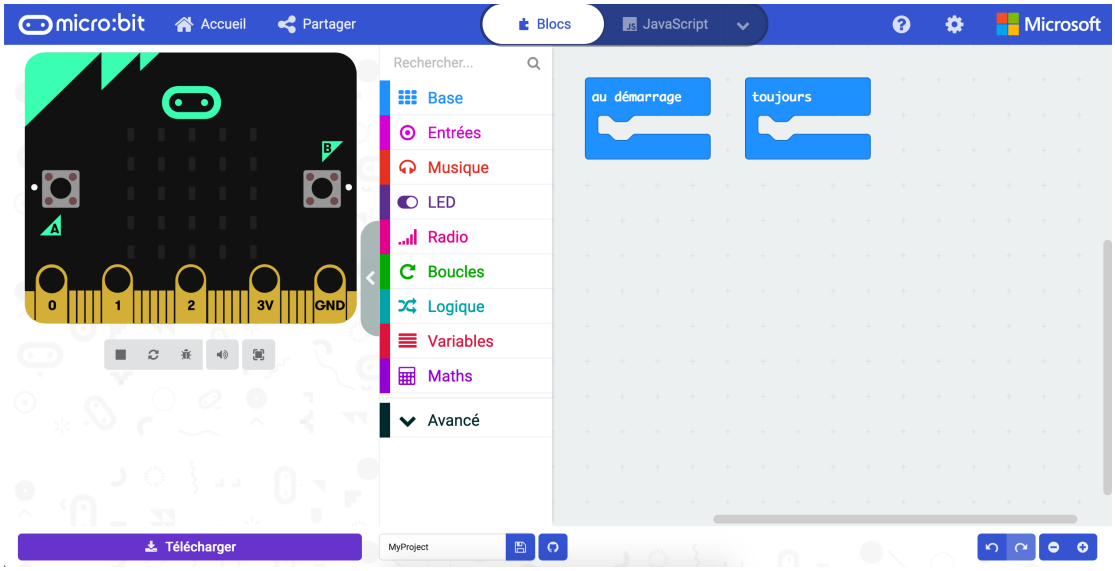


Figure 1.1: L'interface de MakeCode

Scratch est un programme open-source permettant de créer des images, des animations, des histoire, des jeux et de la musique. Il se base sur le principe de block 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.

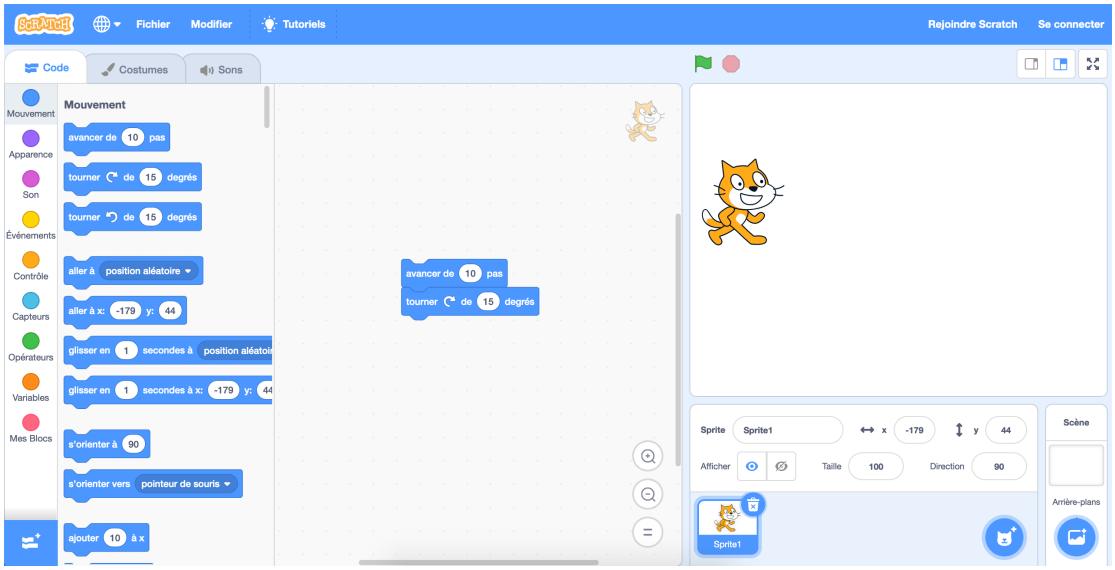


Figure 1.2: L'interface de Scratch

Chapitre 2: FunBlocks

Au début du projet, Funblocks se présentait comme suit:

- Un onglet Edit : 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 block dans une règle ou l'état initial, l'utilisateur peut taper au clavier le label de ce block. Des blocks peuvent être glisser dans un autre block afin de créer un sous-block.
  - Le mode Text est un éditeur de texte. Il permet de représenter les objets utilisés dans le mode Blocks sous forme de code. Le mot clef *init* déclare l'état initial et le mot *case* une règle. Les parenthèse définissent un sous-block.

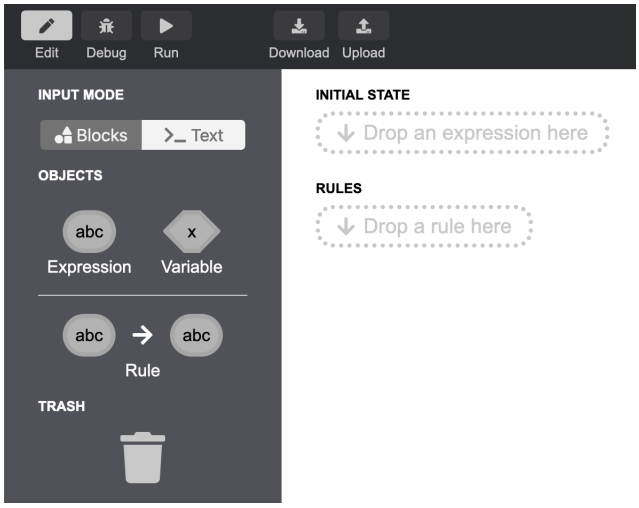


Figure 2.1: Onglet Edit dans le mode Blocks.

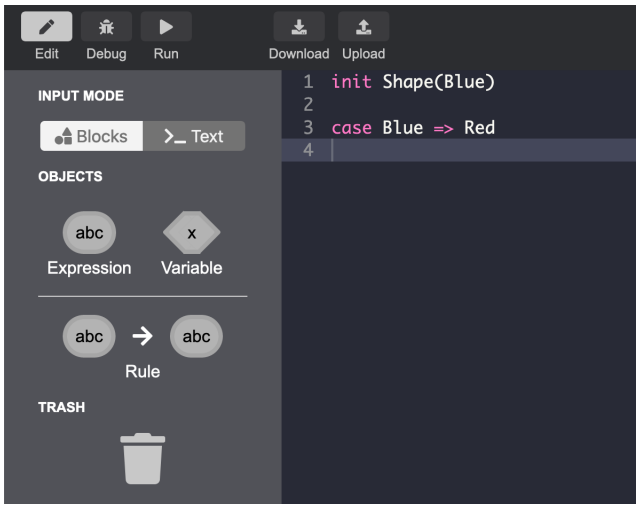


Figure 2.2: Onglet Edit dans le mode Text

- 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 cliquer sur le terme désiré. Un système d'historique permet de revenir à un état précédent (Figure 2.3).
- Un onglet Run : En cours de développement.

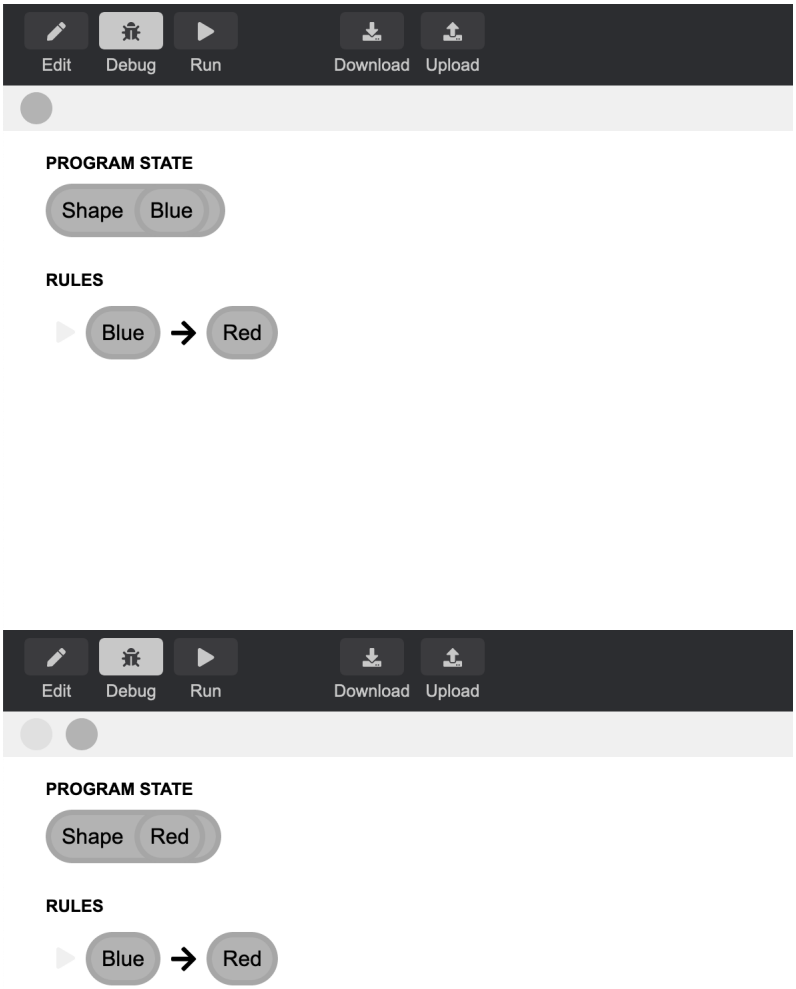


Figure 2.3: Onglet Debug avec l'état avant et après l'application d'une règle

L'état dans Funblock est représenté de trois manières différentes : sous forme de Blocks, 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. 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. Seul 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 Block. 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 appliqué 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 block Blue en block Red.

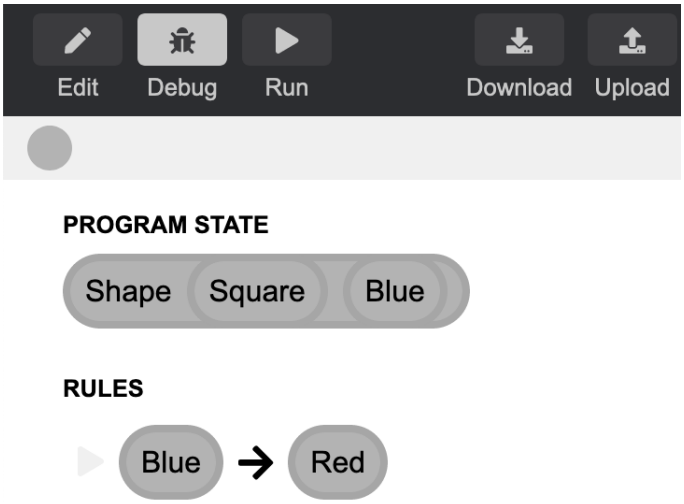


Figure 2.5: Notre programme avec une règle.

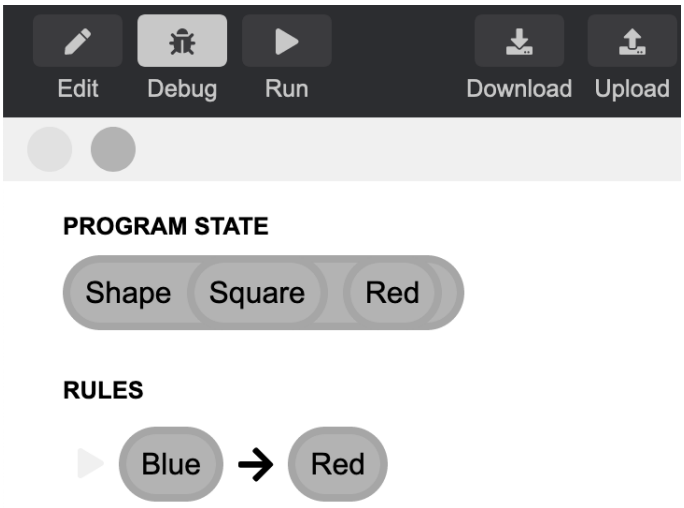


Figure 2.6: Ici, la règle a été appliquée à un sous-terme de notre programme.

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.

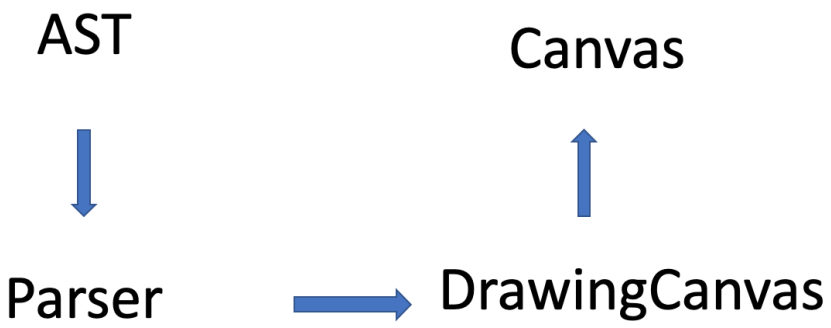


Figure 3.1: Le travail de Funview

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 concept informatique comme des listes. Une grammaire régissant les formes et les positions a du 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.

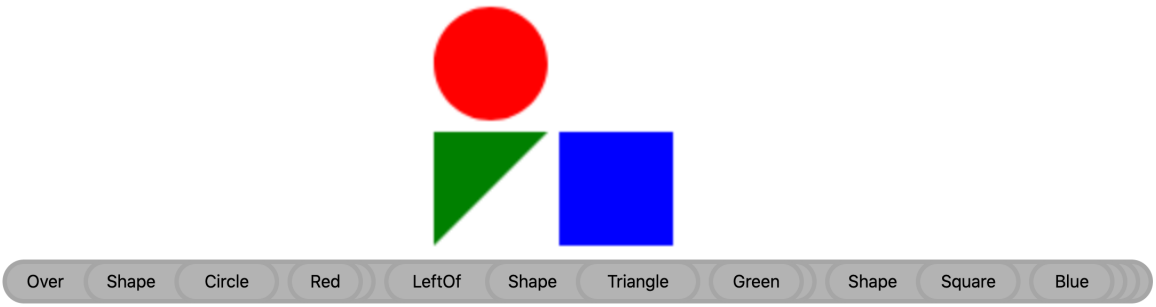


Figure 3.2: Un état accepté par la grammaire sous sa forme graphique et sa forme en block. Le cercle se retrouve au dessus, grâce au terme *Over*, de l'ensemble décrit dans le terme *LeftOf*.

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

<pos> doit avoir une taille deux fois plus
grande que <shapes> afin d'avoir un X,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.

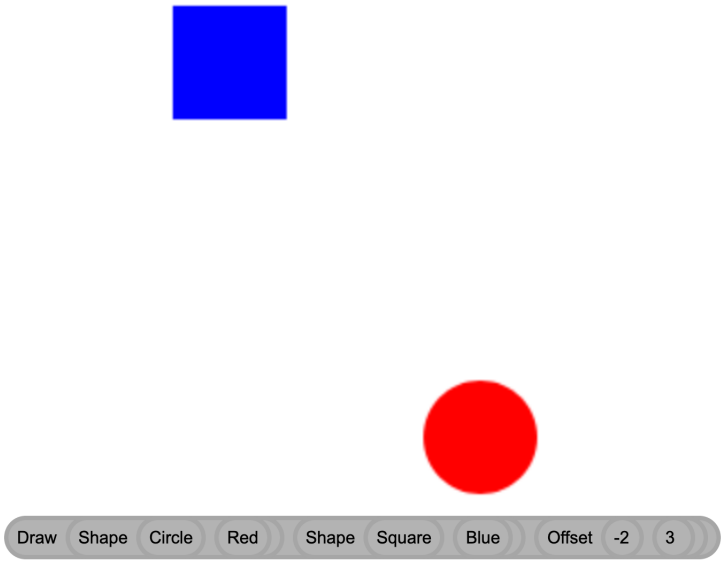


Figure 3.3: Exemple de la commande Draw. Le carré se trouve à la position (-2, 3) par rapport au cercle

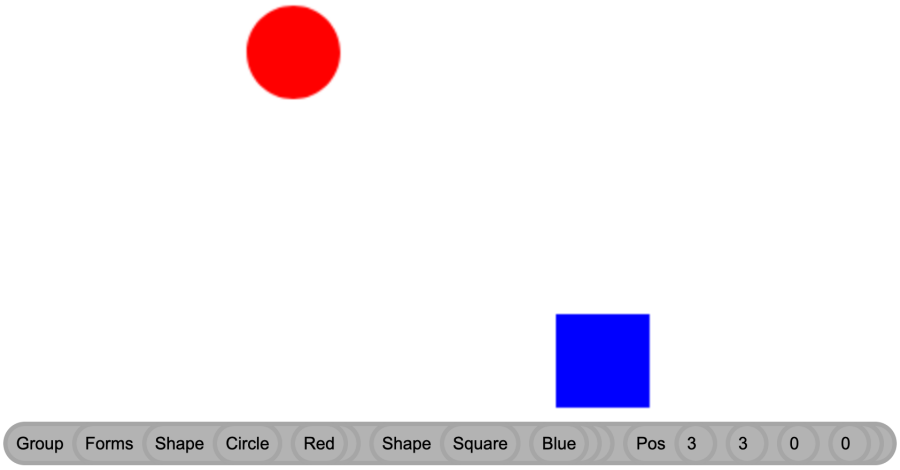


Figure 3.4: Exemple de la commande Group. La position du cercle est (3, 3) et celle du carré est (0,0).

3.2 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 DrawnState. Cette classe contient uniquement la-dite 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 rappellera elle-même selon l'ordre adéquat.

Sur la Figure 3.5, On peut voir la transformation que permet FunView.

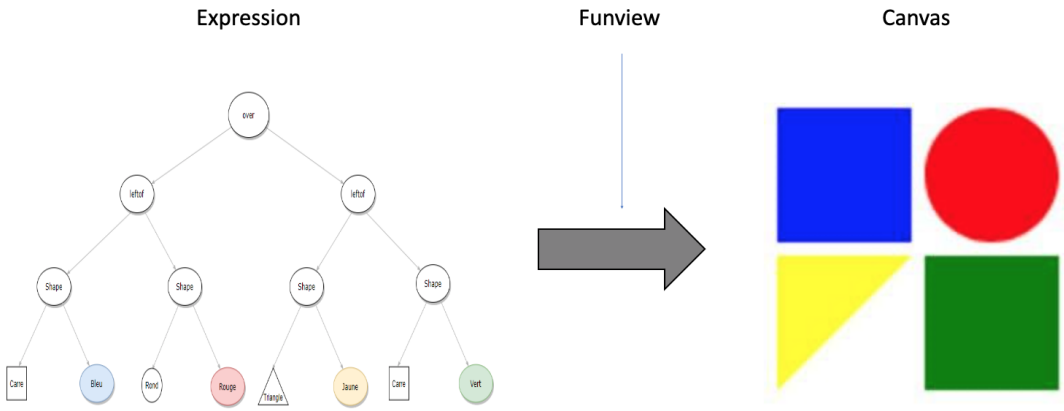


Figure 3.5: Exemple de l'état sous forme AST et sa représentation graphique

La méthode *explore* parcourt l'arbre en profondeur. Dans la figure 3.5, elle commence à la racine *Over* et informe par conséquence 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 3.5. *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 bleu. 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*.

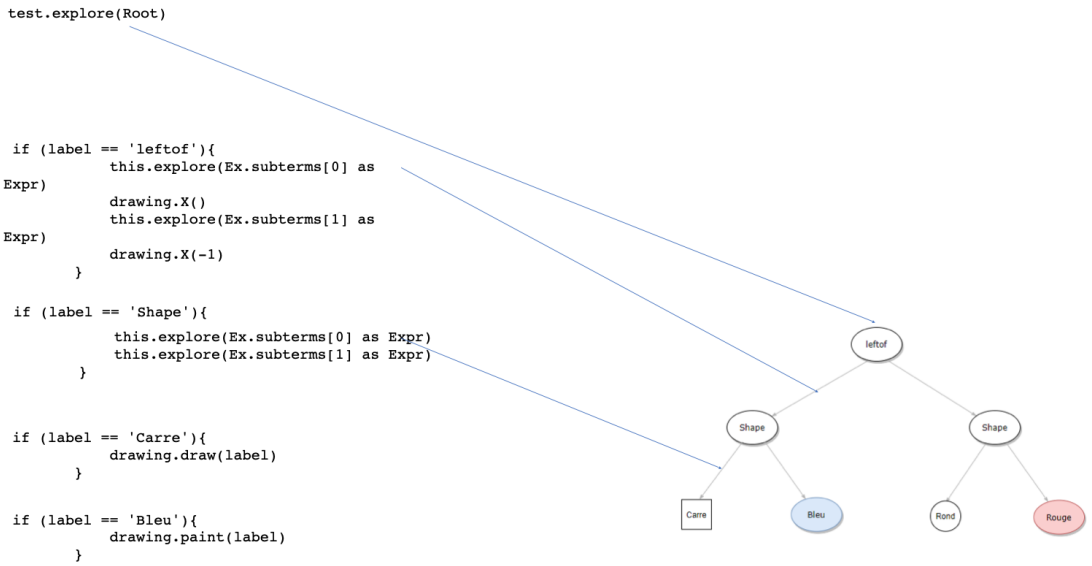


Figure 3.6: Exemple de parcours par la méthode recursive explore

La figure 3.6 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()*.

3.3 Le canvas HTML

Le canvas HTML est un élément 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 [6] 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.

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 designé dans ce but. 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. Cette méthode permet également d'afficher si une erreur a été commise lors de l'écriture des blocks. La méthode *paint* quand à elle assigne une couleur à la forme décrite précédemment. Elle prend également une chaîne de caractères en argument.

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 appelle 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.

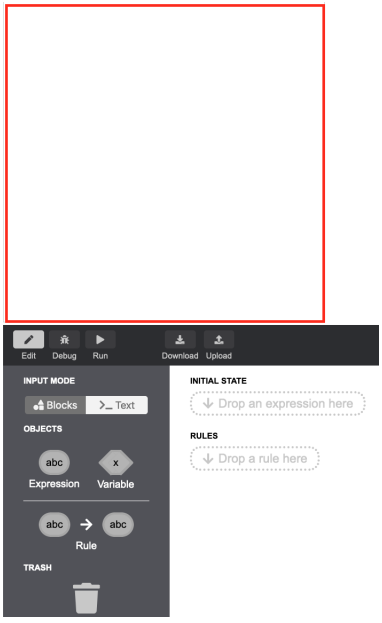


Figure 4.1: On voit ici la zone marquée en rouge réservée au canvas HTML au dessus de l'interface FunBlocks. C'est ici que sera dessiné les formes représentants l'état du programme.

Cette 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 [5] qui permet de créer une application web sur une seule page. La page réagit aux actions performées sur la dite 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.

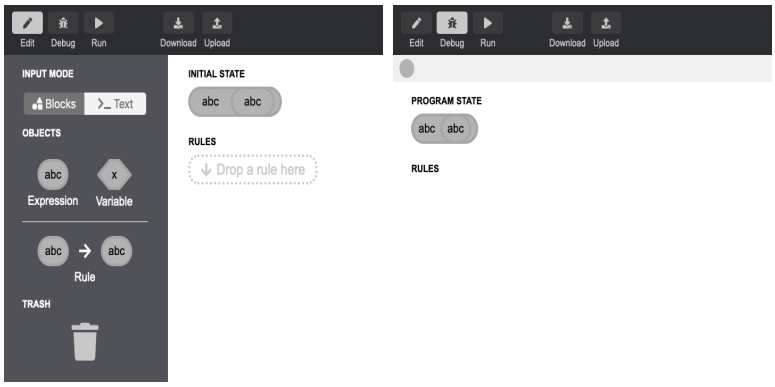


Figure 4.2: Le *Workspace* Edit et le *Workspace* Debug

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 trouve 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 opérationel 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 problème de compréhension.

La grammaire est un apport qui peut être améliorée. 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 seul 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 soucis 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 fichier 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'autre programme tel que MakeCode et Scratch. Il reste cependant bien plus simple et à un stade où beaucoup d'éléments peuvent être rajouté. Durant ce projet il n'a pas été question d'animation, ce que propose MakeCode et Scratch. De plus, les créations graphiques possibles sur ces deux programmes sont poussé 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 block 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] *Cnavas HTML*. URL = https://fr.wikipedia.org/wiki/Canvas$_(HTML)$, consulté en septembre 2020.

[2] MIT. *Scratch*. URL = https://scratch.mit.edu/, consulté en septembre 2020.

[3] Microsoft. *MakeCode*. URL = https://makecode.microbit.org/, consulté en septembre 2020.

[4] Dimitri Racordon. *FunBlocks*. Github = https://github.com/kyouko-taiga/FunBlocks, consulté en septembre 2020.

[5] *React*. URL = https://fr.reactjs.org/, consulté en septembre 2020.

[6] *SVG*. URL = https://fr.wikipedia.org/wiki/Scalable$_V ector_G raphics$, consulté en septembre 2020.