Rapport de projet

Wollenburger Antoine, Chénais Sébastien

Sommaire

Introduction

Il nous a été demandé d'écrire un compilateur, en OCAML, d'un langage de notre cru vers du svg (format pour décrire des images vectorielles). Pour mener à bien cette tâche, nous nous reposerons sur les notions que nous avons vues en cours.

Pour rappel, une image vectorielle est une image dont les éléments sont définis individuellement sous forme d'objet géométrique. Elle se différencie d'une image matricielle qui utilise des pixels. Son principal avantage est qu'elle peut être agrandie à l'infini sans perte de qualité contrairement à une image matricielle. Son inconvénient est que pour atteindre une qualité photo-réaliste, il faut beaucoup de ressource, l'image étant calculée à chaque affichage.

FIGURE 1 – SVG contre Matriciel. fr.wikipedia.org

Chapitre 1

Analyse lexicale et syntaxique de SVGgenerator

Cette section a pour but la définition du langage SVGgenerator et de sa grammaire associée.

1.1 Définition d'une grammaire

La première étape de ce projet a été la création d'un langage et sa grammaire associée. Même si le projet global se fera de manière incrémentale, il faut penser au fonctionnement global dès le début.

1.1.1 Définition d'une image vectorielle

Le fichier SVG est déclaré par l'instruction "drawing" suivi du nom du dessin et la taille du canevas sous la forme "[largeur , hauteur]". Les instructions sont incluses entre les crochets.

Listing 1.1– Définition d'une image vectorielle

```
1 drawing monDessin [256 , 256]
2 {
3     -instructions-
4 }
```

1.1.2 Les instructions

Une instruction peut être de différents types :

Une déclaration de variable : on créé une variable en indiquant son type, son nom et enfin les valeurs qui servent à la construire.

Listing 1.2– Déclaration de variable

```
1 Type ma_Variable( -param- );
```

Le dessin d'une variable : pour les variables dont le type est adéquat, il s'agit de l'instruction qui déclenche l'affichage de ladite variable. La forme est "draw nomDeLaVariable".

Listing 1.3– Dessin d'une variable

```
1 draw ma_Variable;
```

Une boucle : il s'agit d'une boucle itérative. L'instruction est de la forme "For(var=start,end){}". Le bloc entre crochets est répété tant que le compteur n'a pas atteint la valeur "end" (il est à noter que ce compteur, bien que pouvant être utilisé dans le corps de la boucle, doit être déclaré avant la boucle, et que toute modification l'affectant n'influera pas sur le nombre d'étapes de la boucle).

Listing 1.4– Boucle

```
1 Float i(0);
2 For(i=start,end) {
3     - instructions-
4 };
```

Les fonctions et procédures : il s'agit d'un appel a une fonction ou une procédure. L'instruction est de la forme "nomDeLaFonction (-paramètres-)". Pour les fonctions, il est possible de récupérer une valeur de retour.

Listing 1.5– Fonctions et procédures

```
1 ma_fonction (-param-);
```

Quelques instructions n'ont pas encore été implémentées

Voici comment nous aurions procédé le cas échéant :

Une conditionnelle : il s'agit d'un test sur une ou plusieurs conditions qui influe sur la suite de l'exécution. Cette instruction est de la forme "if(-conditions-){}else{}". Si la condition est vérifiée, alors le bloc correspondant au if est exécuté, celui correspondant au else sinon.

Listing 1.6– Conditionnelle

```
1 if( -condition- ){
2     - instructions-
3 }
4 else{
5     -instructions-
6 }
```

Son implémentation nécessite deux choses :
- L'ajout des booléens, et de leurs opérateurs logiques. Cette étape est des plus bénigne, tant ce modèle est proche de celui des expressions arithmétiques.
- L'ajout de la conditionnelle : il eu fallu ajouter les tokens, les règles de parsage, et enfin une action utilisant le if de caml et nos booléens (fonction execute_action_before)

Ajout de propriétés aux objets : On pourrait ajouter de la couleur à l'objet, ou encore la taille du trait... pour ce faire, il suffit d'ajouter un attribut à l'objet Caml, définir ses getters et setters comme pour les attributs classique et ajouter le traitement de ces attributs dans la fonction de génération du code SVG.

Seuls le manque flagrant de temps et la proximité des examens nous ont empêchés de les implémenter.

1.1.3 Les types

Plusieurs types ont été définis à l'origine :

Float : Il s'agit d'un nombre. Son interprétation interne sera le flottant.

Point : Un Point est défini par deux Number correspondant à l'abscisse et l'ordonnée. Sa représentation en SVG est inexistante.

Line : Une Line est définie par deux points correspondant au début et à la fin de la Line. Sa représentation en SVG est une ligne.

Rectangle : Un Rectangle est défini par un Point et deux Number, correspondant respectivement à une des extrémité de la forme, à la longueur et la largeur. Sa représentation en SVG est le rectangle.

Circle : Un Circle est défini par un point et un Number, correspondant respectivement au centre et au rayon de la forme. Sa représentation en SVG est le cercle.

1.1.4 Les commentaires

Le langage autorise les commentaires sous deux formes. Premièrement, le commentaire sur une ligne qui commence par "//". Ce commentaire peut être placé en début ou en fin de ligne. Enfin, le commentaire sur plusieurs lignes encadré par "/*" et "*/". Toutes les instructions commentées ne sont pas interprétées.

```
1 //ceci est un commentaire sur une ligne
2 /* ceci est
3 un commentaire en
4 bloc */
5 -instruction- //commentaire
```

1.1.5 La grammaire

Voici la grammaire associée à SVGgenerator.

Listing 1.7– Grammaire associée a SVGgenerator

```
1  Grammaire SVGgenerator{S,Vt,Vn,R}
2
3  Vt = { 'drawing'; '['; ']'; 'x'; '{'; '}'; 'Point'; '('; ','; ')'; ';'; '
       Line'; 'draw'; 'var'; 'number'; 'eof'; '+'; '-'; '/'; '*'; '.';'='; '
       Rect'}
4  Vn = {S; Bloc{; Instructions; Instr; Declaration; Draw; Function; Param;
       Exp_ar; Var; T; F; Affectation; Subvar}
5
6  R{
7  S                -> Function.Drawing.'eof'
8  S                -> Drawing.'eof'
9  Function         -> 'function'.'var'.'('.Param.')'.Bloc{
10 Function         -> 'function'.'var'.'('.Param.')'.Bloc{.Function
11 Param            -> Param.','.Param
12 Param            -> 'Point'.'var'
13 Param            -> 'Line'.'var'
14 Drawing          -> 'drawing'.'var'.'['.'number' ',' 'number'.']'.Bloc{
15 Bloc{            -> '{'.Instructions.'}'
16 Instructions     -> Instr.Instructions
17 Instructions     -> Instr
18 Instr            -> Declaration
19 Instr            -> Draw
20 Instr            -> Fun
21 Declaration      -> 'Point'.'var'.'('.Exp_ar.','.Exp_ar.')'.';'
22 Declaration      -> 'Line'.'var'.'('.'var'.','.'var'.')'.';'
23 Declaration      -> 'Float'.'var'.'('.'Exp_ar'.')'.';'
24 Declaration      -> 'Rect'.'var'.'('.'var'.','.'Exp_ar'.','.Exp_ar.')'.';'
25 Draw             -> 'draw'.'var'.';'
26 Affectation      -> Subvar.'='.Exp_ar.';'
27 Affectation      -> Subvar.'='.Subvar.';'
28 For              -> 'for'.'('.'var'.'='.Exp_ar.','.Exp_ar.')'.Bloc{
29 Fun              -> 'var'.'('.Var.')'.';'
30 Fun              -> 'var'.'('.Exp_ar.')'.';'
31 Var              -> 'var'.','.Var
32 Var              -> 'var'
33 Exp_ar           -> T.'+'.Exp_ar
34 Exp_ar           -> T.'-'.Exp_ar
35 Exp_ar           -> T
36 T                -> F.'*'.T
```

```
37 | T                -> F. '/' .T
38 | T                -> F
39 | F                -> 'number'
40 | F                -> '(' .Exp_ar. ')'
41 | Subvar           -> Subvar. '.' .'var'
42 | Subvar           -> 'var'
43 | }
```

1.2 L'analyse lexicale

L'analyse lexicale est effectuée au niveau du fichier grapheur_lexer.mll. Nous identifions le vocabulaire terminal Vt tel que déclaré dans la grammaire. C'est a cet endroit que nous gérons les commentaires en ignorant tout entrée entre '/*' et '*/' ainsi que toute entrée entre '/' et une fin de ligne.

De même, nous ignorons tout caractère de type espace, tabulation ou retour charriot.

Un nom de variable est composé d'une lettre puis éventuellement d'une liste de lettre et nombre, quelle que soit la casse.

Un nombre correspond à un flottant de la forme X ou X.X où X est un entier.

Les tokens étant crées, il faut maintenant analyser leur syntaxe.

1.3 L'analyse syntaxique

L'analyse syntaxique à pour objectif la création d'une structure utilisable pour la génération du fichier SVG. Cette structure est un arbre binaire comme défini à la suite. Chaque nœud correspond à une opération qui peut être un token, un nom de variable ou un nombre.

Listing 1.8– Type arbre

```
1  type operation =
2      Drawing
3    | Root
4    | Function
5    | Functions
6    | DrawingSize
7    | BlocEmbrace
8    | Declaration
9    | BlocBrace
10   | BlocPar
11   | Parameters
12   | Parameter
13   | ParametersUse
14   | ParameterUse
15   | FunctionUse
16   | Point
17   | Float
18   | Line
```

```
19      | Rect
20      | Instruction
21      | Comma
22      | Draw
23      | Moins
24      | Plus
25      | Mult
26      | Div
27      | For
28      | Dot
29      | Affectation
30      | Arithm_expr
31      | Var of (string)
32      | Number of (float);;
33
34  type t_arbreB = Empty | Node of node
35          and node = { value: operation; left: t_arbreB; right: t_arbreB };;
```

Voici une représentation visuelle de l'arbre, nœud par nœud. Un nœud de couleur verte signifie qu'il s'agit de la tête de son arbre. Un nœud de couleur rouge représente un nœud non final, à relier à un nœud vert.

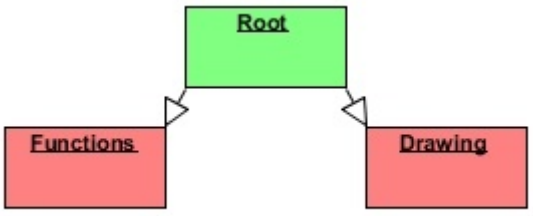

FIGURE 1.1 – Root - tête de l'arbre général.

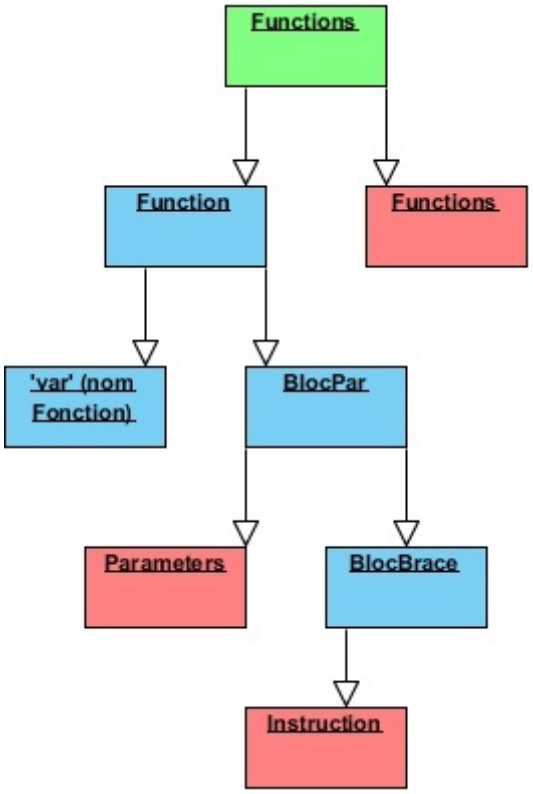

FIGURE 1.2 – Functions - tête pour chaque.

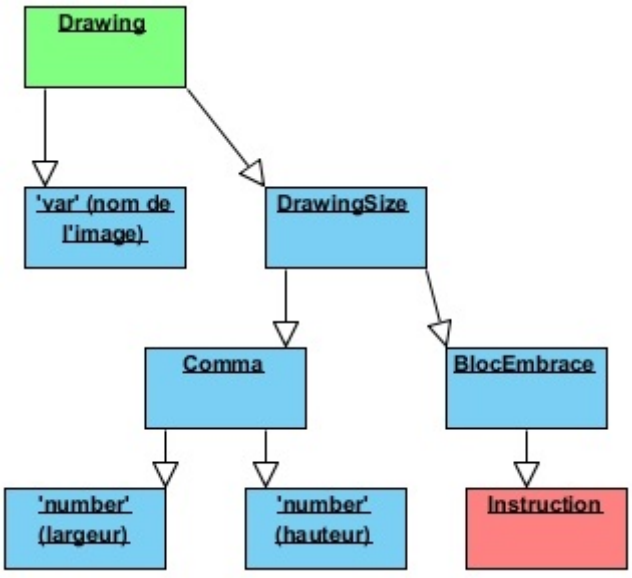

FIGURE 1.3 – Drawing - tête pour le dessin.

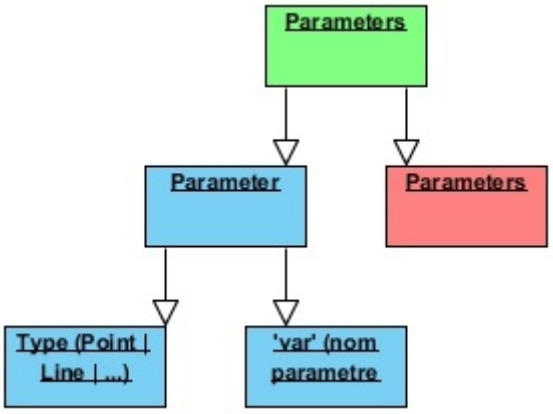

FIGURE 1.4 – Parameters - tête pour une suite de paramètres.

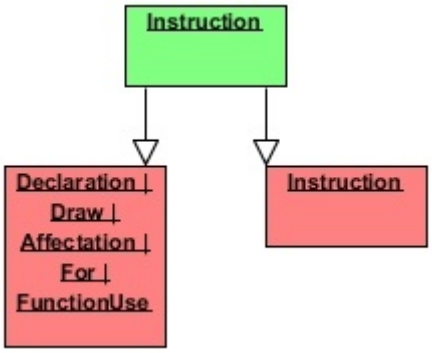

FIGURE 1.5 – Instruction - tête pour une suite d'instruction.

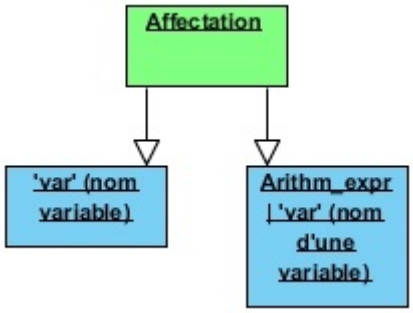

FIGURE 1.6 – Affectation - tête pour une affectation de variable.

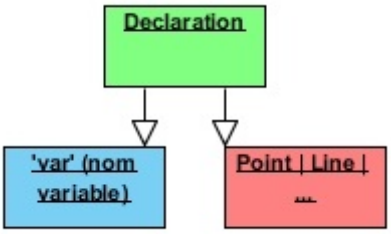

FIGURE 1.7 – Déclaration - tête pour une déclaration de variable.

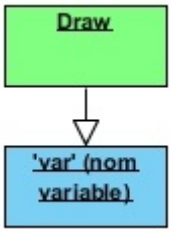

FIGURE 1.8 – Draw - tête pour le dessin d'une variable.

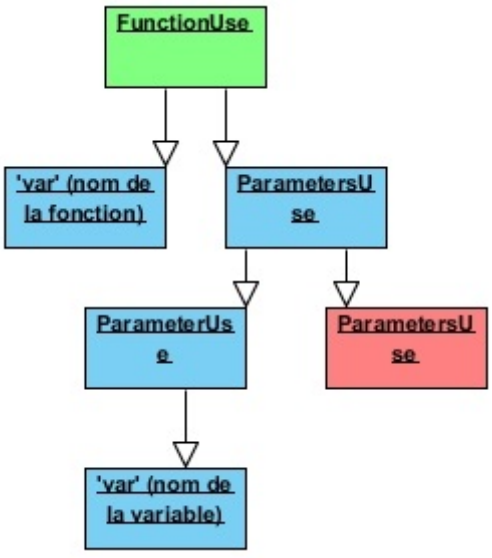

FIGURE 1.9 – FunctionUse - tête pour l'appel à une fonction.

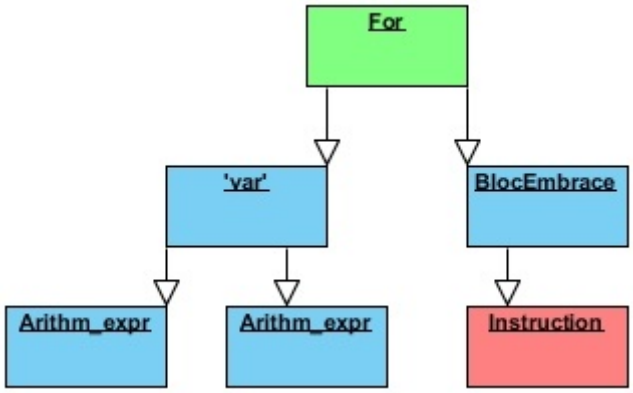

FIGURE 1.10 – For - tête pour une boucle for.

Chapitre 2

Génération du code SVG

2.1 Les types

Voici les définitions des différents types définis.

Listing 2.1– type Float

```
1 class valFloat =
2     object
3       val mutable value = 0.
4       method get_value = value
5       method set_value valp = value <- valp
6     end;;
```

Listing 2.2– type Point

```
1 class point =
2     object
3       val mutable x = 0.
4       val mutable y = 0.
5       method get_y = y
6       method set_y yp = y <- yp
7       method get_x = x
8       method set_x xp = x <- xp
9     end;;;
```

Listing 2.3– type Line

```
1 class line =
2     object
3       val mutable p1 = new point
4       val mutable p2 = new point
5       method get_p1 = p1
6       method set_p1 p1p = p1 <- p1p
7       method get_p2 = p2
```

```
8         method set_p2 p2p = p2 <- p2p
9       end;;
```

Listing 2.4– type Rect

```
1  class rect =
2      object
3        val mutable o = new point
4        val mutable w = 0.
5        val mutable h = 0.
6        method get_o = o
7        method set_o p1p = o <- p1p
8        method get_w = w
9        method set_w p2p = w <- p2p
10       method get_h = h
11       method set_h p2p = h <- p2p
12     end;;
```

Chaque type est associé avec un wrapper pour plus de commodité :

Listing 2.5– Wrapper

```
1  type all_types = Point_wrap of point | Line_wrap of line | Float_wrap of
       valFloat | Rect_wrap of rect;;
```

2.2 Les variables

2.2.1 affectation

Pour être utilisée dans la suite du programme, une variable doit être déclarée. Dans notre cas, la déclaration s'accompagne aussi d'une affectation. Il est par conséquent impossible de crée une variable nulle. Les variables sont stockées au fur et à mesure de leur apparition dans une hash table, d'une taille arbitraire de 1000. La clé est donc le nom ce qui peut poser problème lors de l'utilisation des fonctions. Pour y remédier, lorsqu'on sort d'un domaine de visibilité (représentés par un bloc de {}), les variables déclarées dans ce bloc sont supprimées. Ainsi, les variables déclarée dans le bloc général (drawing) sont globales par rapport aux fonctions. Si une fonction possède une variable du même nom, la variable de la fonction sera utilisée. La valeur associée à la clé est le type de la variable. Si une même variable est déclarée deux fois, le compilateur génère une erreur.

2.2.2 réutilisation

Lors de l'appel à une variable, sa valeur est recherchée dans la table. Si la valeur est présente et le type adéquat à la situation, l'exécution se poursuit. Sinon, le compilateur génère une erreur. Pour les types primitifs, ici Float, la variable est remplacé par sa valeur.

2.2.3 les objets

En dehors du Float, chaque type de variable correspond à un objet (Rect, Point,...). Ces objets ont des attributs de type primitif ou objet. Il est possible d'atteindre ces attributs à l'aide du '.' (point). Voici ces attributs :

Point : x et y de type Float.

Line : p1 et p2 de type Point.

Rectangle : o de type Point et w,h de type Float.

Ex : monPoint.x ;
L'affectation et la réutilisation des attributs est possible.

2.3 Les opérations arithmétiques

Cette section va être succincte, les opérations arithmétiques étant un cas d'étude fréquent.

Elles sont faisable partout excepté lors de l'appel à une fonction. En effet, la fonction effectue une vérification de type avant de s'exécuter, et le type d'une opération arithmétique n'est pas Float tant qu'on ne la pas résolue. C'est un problème à régler.

2.4 les fonctions

Les fonctions sont définies avant le bloc drawing, afin que les noms soient déjà connus. Le code associé à une fonction se trouve sur la branche gauche de l'arbre, un parcours est donc faisable sans avoir besoin de stocker les références des fonctions. A chaque appel de fonction, l'appel est remplacé par le code dans l'arbre. Le nom des variables est modifié en conséquence si besoin est.

Chapitre 3

Le langage, spécificités, contraintes

Annexe A

Journal des évolutions

A.1 Lexer

La seconde étape, après avoir déterminé le langage, fut d'écrire le lexer. Rien de bien difficile. De base nous n'avons intégré que les tokens primordiaux.

A.2 Parser : vers un arbre pour représenter le fichier

Cette étape, bien qu'un peu plus complexe fut rapidement menée à bien. Il est à noter que lexer et parser continuèrent d'évoluer à chaque ajout de vocabulaire visant à enrichir le langage.

A.3 Interlude : table S-R et inutilité

A ce stade nous avons commencé à écrire la table shift-reduce correspondant à notre grammaire, avant de nous rendre compte de son inutilité à ce stade...

A.4 De l'exploitation de l'arbre

A.4.1 Lecture du drawing

Très tôt, un système "d'exécution de l'abre" fut écrit, ceci afin de pouvoir voir les résultats de notre dur labeur. La toute première action implémentée fut le "drawing", pour créer un fichier svg.

A.4.2 Déclaration de procédures

Vint ensuite la déclaration de procédures, avec un nombre non fixé de paramètres bien entendu. Très utile... enfin pas à ce moment,car déclarer une procédure ne la rend pas utilisable.

A.4.3 La déclaration de variables

Dans la volée nous avons enchainé avec la déclaration des variables : à ce stade elles étaient toutes stockées dans une table globale, qui contenait leur valeur (Hashtbl (string,all_types)).

A.4.4 De l'utilisation des procédures

Pour utiliser une procédure, deux étapes sont nécessaires :
- La première est de reconnaitre l'appel à une procédure. Il fallu donc identifier les tronçons d'arbres s'en occupant.
- La seconde est de recopier le code de la procédure aux endroits de ses appels, tout en renommant les variables correctement.

A.4.5 Principe d'actions

Quelques actions firent leur apparition lors des étapes précédentes. La déclaration d'une variable et le draw en font partie. Nous pouvions alors utiliser des procédures basques pour dessiner les formes implémentées (lignes et points).

A.4.6 Les 0.

Un petit ennui du côté de la génération du svg nous apparut : tous les navigateurs internet, mis à part firefox, n'arrivaient pas à afficher les svg contenant des floats se terminant pas un '.'. Nous résolûmes donc ce problème.

A.5 Le boulversement FOR

A.5.1 L'idée du for

En substance, le for, lors de l'exécution du code, consiste à répéter ce code un certain nombre de fois.

A.5.2 Ajout des floats

Pour pouvoir incrémenter le compteur du for, nous introduisîmes les floats. Sans, le for fonctionnait, mais on ne pouvait pas utiliser le compteur de boucle... pas très utile de recopier n fois là même forme.

A.5.3 L'arrivée de la notion de visibilité

A ce moment, nous avons décidé d'implémenter la notion de visibilité pour les blocs entre '{' et '}'. l'utilité ? Pouvoir déclarer en dehors d'un for une variable déjà déclarée dans le for. (à noter que celà ne fonctionne pas pour les procédure, enfin pas quand la déclaration se fait avant son appel : en effet, chaque procédure hérite des variables de drawing).

A.5.4 Réunification du préparsage et de l'exécution

Pour pouvoir mettre en place la notion de visibilité, les variables ne devaient plus êtres ajoutées lors d'une étape préliminaire à la table des variables, mais cette table devait se mettre à jour dynamiquement. Ainsi, le préparsage perdit son utilité.

A.5.5 La réalité du FOR

Tout ceci mis en place, le for pu enfin être implémenté.

A.6 Interlude : liens entre objets

Quelques couacs subsistent sur les liens entres objets. En effet, une ligne utilisant un point ne se verra pas toujours changer quand ce point change.

A.7 Opérations sur les objets

Cette étape, assez prise de tête, consista en la mise ne place (parsing + interprétation) des "variables.opération". Ceux-ci étant récursifs (comme dans "ligne.p1.y").

A.7.1 Récupération de la valeur

La première étape fut de pouvoir récupérer les valeurs des sous-composants des objets, par exemple pour les réaffecter.

A.7.2 Affectation d'une valeur

En second vint la modification des sous-composants des objets (par exemple du premier point d'une ligne).

A.8 L'ajout d'objets, simplicité

A.8.1 Déclaration

Il faut d'apport modifier lexer et parser pour pouvoir les lires. Ensuite, il est nécessaire d'ajouter notre objet à la fonction qui gère les déclarations et les ajouts à la table de variables.

A.8.2 Opérations

On peut si necessaire, ajouter des opérations sur nos objets. Il suffit de modifier les méthodes permettant de get et de set ces valeurs (prendre exemple sur les autres).

A.8.3 Dessin

Là, rien de plus simple : modifier l'action placée sur le draw pour écrire une ligne de svg utilisant notre objet.