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 vu 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.

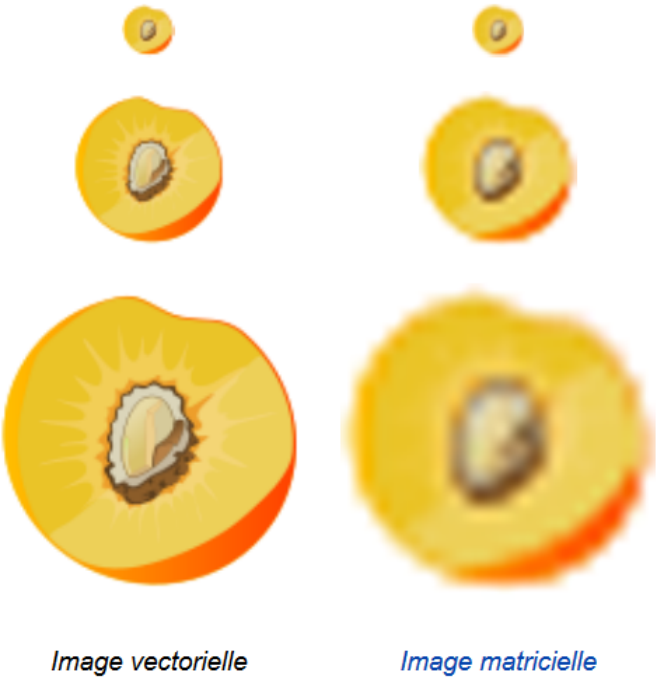

FIGURE 1 – SVG contre BMP. fr.wikipedia.org

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

```
if( -condition- ){
    - instructions-
}
else{
    -instructions-
}
```

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 :

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.

```
//ceci est un commentaire sur une ligne
/* ceci est
un commentaire en
bloc */
-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'; '+'; '-'; '/'; '*'}
4  Vn = {S; Bloc{; Instructions; Instr; Declaration; Draw; Function; Param;
       Exp_ar; Var; T; F}
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 Draw             -> 'draw'.'var'.';'
24 Fun              -> 'var'.'('.Var.')'.';'
25 Fun              -> 'var'.'('.Exp_ar.')'.';'
26 Var              -> 'var'.','.Var
27 Var              -> 'var'
28 Exp_ar           -> T.'+'.Exp_ar
29 Exp_ar           -> T.'-'.Exp_ar
30 Exp_ar           -> T
31 T                -> F.'*'.T
32 T                -> F.'/'.T
33 T                -> F
34 F                -> 'number'
35 F                -> '('.Exp_ar.')'
36 }
```

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

```
type operation =
    Drawing
  | Root
  | Function
  | Functions
  | DrawingSize
  | BlocEmbrace
  | Declaration
  | BlocBrace
  | BlocPar
  | Parameters
  | Parameter
  | ParametersUse
  | ParameterUse
  | FunctionUse
  | Point
  | Float
  | Line
  | Instruction
  | Comma
  | Draw
  | Moins
  | Plus
  | Mult
  | Div
  | For
  | Affectation
  | Arithm_expr
  | Var of (string)
  | Number of (float);;

type t_arbreB = Empty | Node of node
        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.

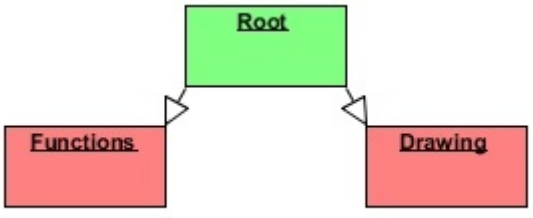

FIGURE 1.1 – Root - tête de l'arbre général.

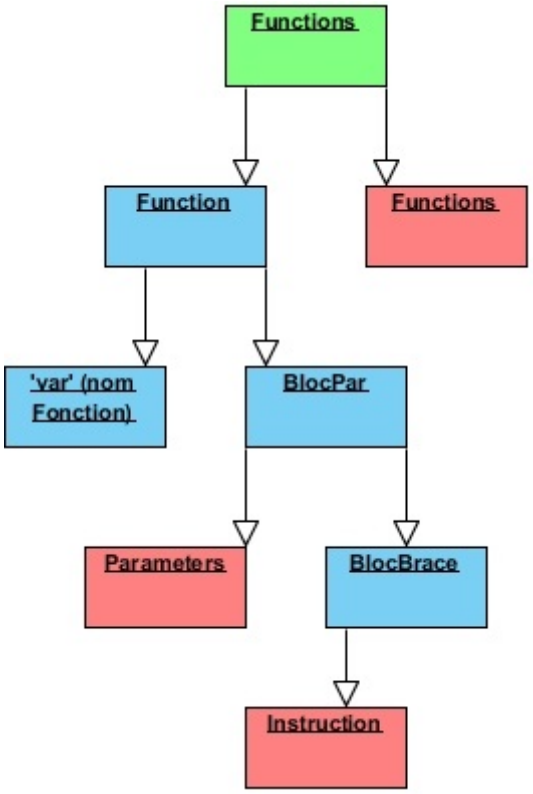

FIGURE 1.2 – Functions - tête pour chaque.

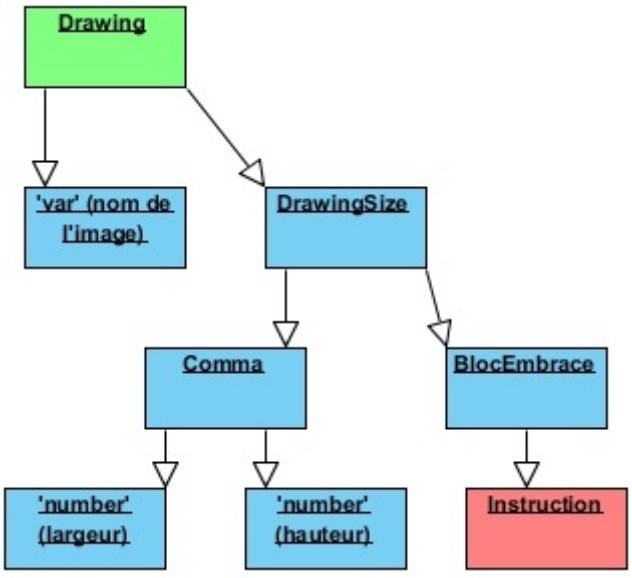

FIGURE 1.3 – Drawing - tête pour le dessin.

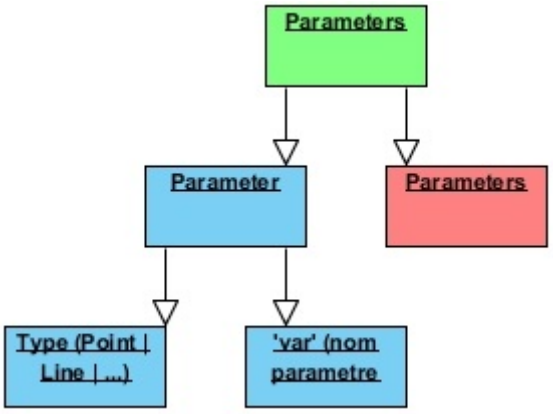

FIGURE 1.4 – Parameters - tête pour une suite de paramètres.

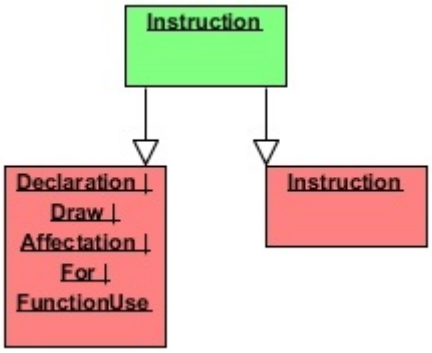

FIGURE 1.5 – Instruction - tête pour une suite d'instruction.

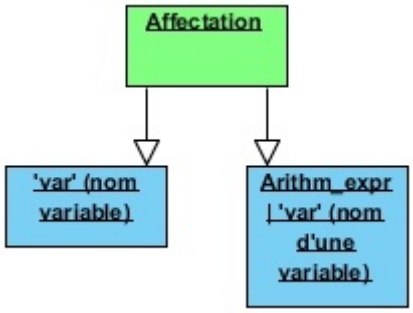

FIGURE 1.6 – Affectation - tête pour une affectation de variable.

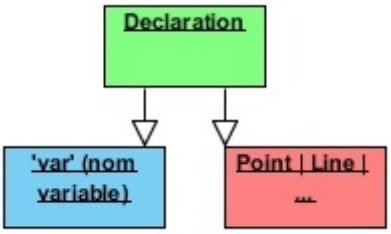

FIGURE 1.7 – Déclaration - tête pour une déclaration de variable.

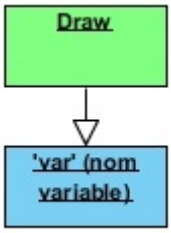

FIGURE 1.8 – Draw - tête pour le dessin d'une variable.

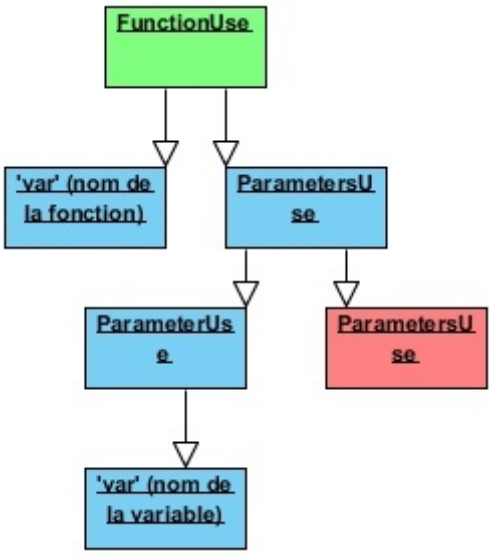

FIGURE 1.9 – FunctionUse - tête pour l'appel à une fonction.

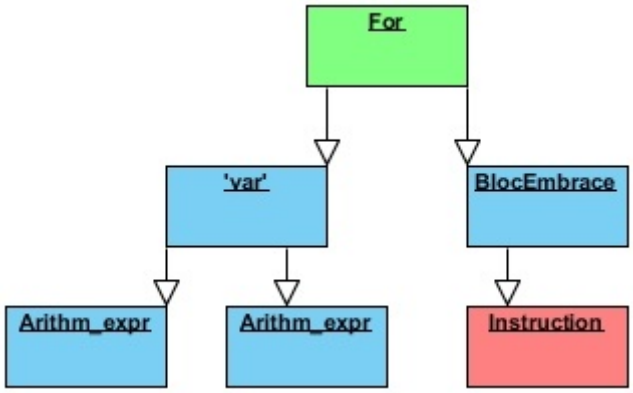

FIGURE 1.10 – For - tête pour une boucle for.

Chapitre 2

Transformation de l'arbre binaire

2.1 L'appel d'eval_val

Cette fonction vérifie la syntaxe de l'arbre binaire donné en entrée. Par la même occasion, elle recopie le code de chaque procédure partout où elle est appelée.

2.2 L'appel d'execute_the_code

Cette fonction parse l'arbre tout en gérant les variables, leurs affectations, leurs déclarations, et effectue des action sur certains tokens bien particuliers. Ainsi un tolen "Draw" déclenchera un print_endline. De même pour "Drawing". Quant à "For", il déclenche l'appel d'un for de caml qui exécute n fois le code présent dans le for, en mettant à jour la variable incrémentale.

2.3 Gestion des déclarations

2.3.1 Déclaration de variables

2.3.2 Déclaration de fonctions

2.4 Gestion de l'accès aux propriétés des objets

2.4.1 Affectaion

2.5 Gestion de la recopie de code pour l'exécution des procédures

2.6 Notion de visibilité pour les variables