

# Rapport de projet

WOLLENBURGER Antoine, CHÉNAIS Sébastien

# Sommaire

|   |          |
|---|----------|
| <b>Introduction</b>                                     | <b>1</b> |
| <b>1 Analyse lexicale et syntaxique de SVGgenerator</b> | <b>3</b> |
| 1.1 Définition d'une grammaire . . . . .                | 3        |
| 1.1.1 Définition d'une image vectorielle . . . . .      | 3        |
| 1.1.2 Les instructions . . . . .                        | 3        |
| 1.1.3 Les types . . . . .                               | 4        |
| 1.1.4 Les commentaires . . . . .                        | 5        |
| 1.1.5 La grammaire . . . . .                            | 5        |
| 1.2 L'analyse lexicale . . . . .                        | 6        |
| 1.3 L'analyse syntaxique . . . . .                      | 6        |
| 1.4 Et le bonheur arrive. . . . .                       | 7        |

# 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](http://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 crée 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 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 1.4– Conditionnelle

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

**Une boucle** : il s'agit d'une boucle sur condition. L'instruction est de la forme "while(-conditions-){}". Le bloc entre crochets est répété jusqu'à ce que la condition soit invalidée.

## Listing 1.5– Boucle

```
1 while( -condition- ){
2     -instructions-
3 }
```

**Les fonctions et procédures** : il s'agit d'un appel à 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 1.6– Fonctions et procédures

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

### 1.1.3 Les types

Plusieurs types ont été définis à l'origine :

**Number** : 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'; '+'; '-'; '/'; '*' }
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 à 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 ou X est un entier.

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

## 1.3 L'analyse syntaxique

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

## 1.4 Et le bonheur arrive.

Viens alors la partie de loin la plus fastidieuse. En effet, pour transformer cet arbre binaire en svg, nous avons besoin d'une implémentation d'un automate à pile. Pour ceci point de choix. Nous passerons par une table shift-reduce. A la vue de la taille du vocabulaire, cette table est tout sauf petite.