Rapport de projet : Dessine moi un Bison

Nicolas Endredi, Baptiste Oruezabal

10 mai 2013

Table des matières

1 Présentation du projet

1.1 Analyse lexicale et syntaxique

Écrire un programme implique d'utiliser un langage particulier qui possède sa propre syntaxe. Lors de la compilation du programme, il nous faut donc vérifier que tous les mots du langage soient reconnus et que les phrases soient correctement définies. Nous devons donc analyser lexicalement notre code puis syntaxiquement. Notre programme est jugé correct si aucune erreur n'est soulevée lors de cette vérification.

1.2 Librairie de dessin : Cairo

La bibliothèque Cairo permet de dessiner sur une surface qui pourra être aussi un fichier (pdf, ps, png, etc...) ou une fenêtre (GTK, Quartz, etc...).

1.3 Analyse lexicale : Flex

Le programme Flex permet d'analyser son entrée standard et détecte des mots clés que le programmeur a définit. On peut donc agir sur le programme lorsqu'on reconnaît un certain mot.

1.4 Analyse syntaxique : Bison

Le programme Bison, jeu de mot sur l'utilitaire Yacc, permet quant à lui de vérifier qu'une phrase est bien écrite. Il se sert d'un fichier de grammaire afin de savoir comment doivent être écrites les différents types de phrases.

1.5 Sujet du projet

Le but du projet est d'élaborer et d'implémenter un langage de programmation de dessin vectoriel en utilisant la bibliothèque graphique Cairo.

2 Architecture logicielle

Nous avons choisi de répartir nos fichiers selon leur fonction dans le projet. Ainsi, les ressources sont isolées du code c ou des fichiers binaires (produits lors de la compilation) par exemple. Voici à quoi sert chaque dossier (on se place à la racine du projet).

- **src** : contient le code source de tout le projet.
  - **bison** : contient notre analyseur syntaxique
  - **c** : contient nos structures de données, le modèle
    - **cairo** : définit la structure "surface"
    - **chemin** : définit la structure "chemin"
    - **image** : définit la structure "image"
    - **listes** : définit les structures de toutes les listes (de points, de chemins, d'images)
    - **points** : définit les différents types de points (cartésien, polaire et point générique)
  - **flex** : contient notre analyseur lexical
  - **includes** : contient tous les fichiers d'en-têtes pour notre projet (reprend l'arborescence du dossier c
  - **tests** : contient tous nos fichiers de tests unitaire avec la même arborescence que le dossier c
- **bin** : contient les fichiers générés lors de la compilation
- **ressources** : contient tous les fichiers qui peuvent être utilisés ou générés par l'exécutable (un fichier de commandes d'exemple, le fichier pdf généré par cairo, etc...)

Pour compiler notre projet, nous avons choisi d'utiliser l'utilitaire CMake. C'est pourquoi, un fichier CMakeLists.txt existe dans le répertoire racine du code source et celui des tests. La compilation se déroule en deux temps avec les commandes suivantes.

1. **cmake** : (dans un dossier hors répertoire src/) génère les différents makefile et les fichiers de configuration
2. **make** : lance effectivement la compilation

3 Fonctionnalités

3.1 Commandes reconnues

Nous avons défini qu'une instruction valide était obligatoirement terminée par le caractère " ;" comme en c. La liste des commandes reconnues sont : draw, fill, img et var. Elles sont expliquées ci dessous.

3.2 Lignes brisées

Pour dessiner une ligne brisée, nous stockons sa couleur, son épaisseur et la liste des points appartenant à la ligne dans un chemin. Ainsi, cela nous permet de définir une bonne fois pour toutes les données concernant un chemin.

Les points reconnus sont les points cartésiens et les points polaires. Comme nous ne manipulons que des points cartésiens, tout point polaire détecté est automatiquement convertit en son équivalent cartésien. Cette conversion est assurée par la structure Point et permet ainsi une abstraction : le chemin manipule une liste de points qu'ils soient polaire ou cartésien dans la ligne de commande.

Notre grammaire reconnaît deux types de coordonnées :
- **simples** : nombre directement utilisables
  Exemple : (1,2)
- **complexes** : résultat d'une expression mathématique à calculer
  Exemple : $((7 * (6 * 8))/(772/2)$, $((2 * (5 + 6))/11)) \Leftrightarrow$ (1,2)

Pour créer un chemin de n points, on utilise la commande
draw (P1) – (P2) – ... – (Pn).

Nous pouvons également définir des cycles grâce au mot clé **cycle** qui remplace la définition d'un point. Ce mot clé sera alors remplacé par le premier point du chemin considéré. La commande **draw (P1) – cycle – ... – (Pn)** sera donc équivalente à celle ci **draw (P1) – (P1) – ... – (Pn)**.

3.3 Remplissage de formes

La commande draw permet de tracer un chemin sans pour autant remplir la forme ainsi dessinée. Au contraire, la commande fill a pour rôle de dessiner le chemin et de remplir la forme. Notre grammaire reconnaît la syntaxe suivante : **fill (P1) – (P2) – ... – (Pn)**
Le chemin ainsi créé aura une variable membre positionnée à 1 et saura qu'il devrait être rempli et non seulement dessiné.

3.4 Utilisation des variables

Lorsque nous avons commencé à définir la couleur et l'épaisseur d'un chemin, il nous a semblé important de les définir dans des variables. Nous avons donc modifié notre grammaire pour qu'elles soient reconnues. Une variable est reconnue si elle suit la logique suivante : **var type ID = value ;** Nous maintenons une liste de variables globales qui ont été créées par cette commande. Lorsqu'une variable est reconnue lexicalement et syntaxiquement correcte, on l'ajoute à la liste. Etant donné que l'on

peut stocker plusieurs types dans nos variables, nous avons choisi de définir une structure variable. Ainsi le type, le nom et la valeur de la variable sont sauvegardés.

3.5 Images

Les images sont une liste d'instructions qui ne sont pas exécutées immédiatement. Comme les commandes draw et fill retournent un chemin, les images ne sont en réalité qu'une liste de chemins. En outre, une image peut très bien contenir d'autres images ou des variables. De plus, toutes les variables déclarées dans un bloc image seront, à terme, locales à ce bloc (pour l'instant, elles sont globales). Exemple de commande pour une image :

img{ var point p1 = (1,2) ; var point p2 = (2,3) ;
draw (p1) – (p2) ;
img{ [autres instructions]} ; } ;

Les deux images sont imbriquées et les points ne seront accessibles que dans ces deux images.

4 Problèmes rencontrés

Au cours de ce projet, nous avons rencontré des problèmes à propos de
- la grammaire
- la gestion des points
- les variables.

En effet, nous avons longuement travaillé sur l'écriture de la grammaire et nous avons été obligés de la remanier à chaque nouvelle implémentation des chemins, images ou autres fonctionnalités.
D'autre part, les coordonnées des points ne sont pas correctement gérées lorsqu'elles sont exprimées dans des expressions mathématiques : les calculs semblent erronés, mais nous n'avons toujours pas trouvé pourquoi. Ceci a pour conséquence que les "points relatifs" exprimés sous la forme "– +(x,y)" ne peuvent pas être gérés.
Enfin, les variables n'ont été conçues que comme variables globales et, bien qu'elles soient lexicalement et syntaxiquement reconnues, elles ne sont pas gérées dans le code. On ne peut donc pas définir de variables dans notre code, pour l'instant

5 Conclusion

La majorité des fonctionnalités demandées sont traitées au moins dans la grammaire (quelques unes présentent des problèmes). Le code rendu est lisible et globalement testé (quelques erreurs d'exécution).
De nombreuses améliorations sont possibles mais nous laissons cela au bon vouloir de développeurs souhaitant reprendre notre travail.