Rapport de projet d'Analyse Syntaxique : **langage de programmation de dessins vectoriels**

Salmane Bah & Benjamin Chalaux & Timothée Sollaud
Université de Bordeaux 1

10 mai 2013

Table des matières

1 Présentation générale du projet

Dans le cadre de l'Unité d'Enseignement "Analyse Syntaxique", suivie durant la formation de Licence 3 informatique à l'université de Bordeaux 1, il nous a été proposé de définir un langage de programmation de dessins vectoriels dont la compilation génère du code C s'appuyant sur la bibliothèque "Cairo" pour effectuer le tracé du dessins.

Nous avons appélé le langage "veclang", il a typage statique avec portée lexicale, le compilateur du langage a été appelé vlcc. un exemple d'utilisation de celui-ci est :

```
./vlcc demo.vl
```

1.1 Les fichiers sources

- veclang-grammar.y & veclang-lexer.l : Il s'agit respectivement de la grammaire et du flex permettant d'analyser le fichier à compiler.
- ast.c/h : Ce sont les fichiers sources pour la création et la modification de l'arbre syntaxique créé par la grammaire et utilisé pour la génération de code.
- scope_management.c/h & symtab_management.c/h : Il s'agit des fichiers sources permettant de gérer les différents contextes et leurs variables.
- queue_list.c queue.h : Ce sont les fichiers source implémentant une file.
- code_generation.c/h : Il s'agit des fichiers sources permettant de générer le code C correspondant à l'arbre syntaxique précédemment généré.
- vlcc.c : Il s'agit du compilateur. Il réalise successivement toutes les étapes permettant de générer le code C correspondant au fichier à compiler passé en paramètre.
- demo.vl : Il s'agit d'un exemple de fichier pouvant être compilé.

Pour lancer le programme il suffit d'effectuer un "make all" qui compilera tout les fichiers requis, puis un "make draw" qui exécutera le code présent dans demo.vl et vous montrera le dessin obtenue dans output.pdf, le code généré étant présent dans demo.c.

1.2 Les spécifications du langage

1.2.1 Les variables

Les déclarations :

scal nomScal1, nomScal2 ... ;
path nomPath1, nomPath2 ... ;
pict nomPict1, nomPict2 ... ;

Les affectation :

nomScal := *number* ;
nomPath := (*number*, *number*)--(*number*,*number*) ... ;
nomPict := image{ *instructions* } ;

number correspond à une expression arithmétique composée d'entier et de variables de type scal.

1.2.2 Les commandes

- draw(fill) (*number*, *number*)--+(*number*, *number*) ... ;
- draw(fill) (*number* : *number*)--cycle ... ;
- draw(fill) (*number* : *number*)--*var* ; (avec *var* de type path)
- draw(fill) *var* ; (avec *var* de type path)
- draw *var* ; (avec *var* de type pict)
- line_width *number* ;
- line_color *number number number* ; (avec *number*, des entiers entre 0 et 100, représentant les couleurs rouge vert bleu)

number correspond à une expression arithmétique composée d'entier et de variables de type scal.

1.2.3 Les fonctions

- var := translate ($transformable$, $point$) ; renvoi $transformable$ translaté de vecteur $point$.
- draw translate ($transformable$, $point$) ; dessine l'objet translaté.
- var := rotate($transformable$, $point$, $angle$) ; renvoi $transformable$ après rotation de centre $point$ et d'angle $angle$.
- draw rotate($transformable$, $point$, $angle$) ; dessine l'objet après rotation.

$transformable$ peut être un objet de type path ou pict.

1.2.4 Les conditionnelles

if ($condition$) { $instructions$ } ;
if ($condition$) { $instructions$ } else { $instructions$ } ;

1.2.5 Les boucles

for ($assignement$; $condition$; $assignement$){ $instructions$ } ;

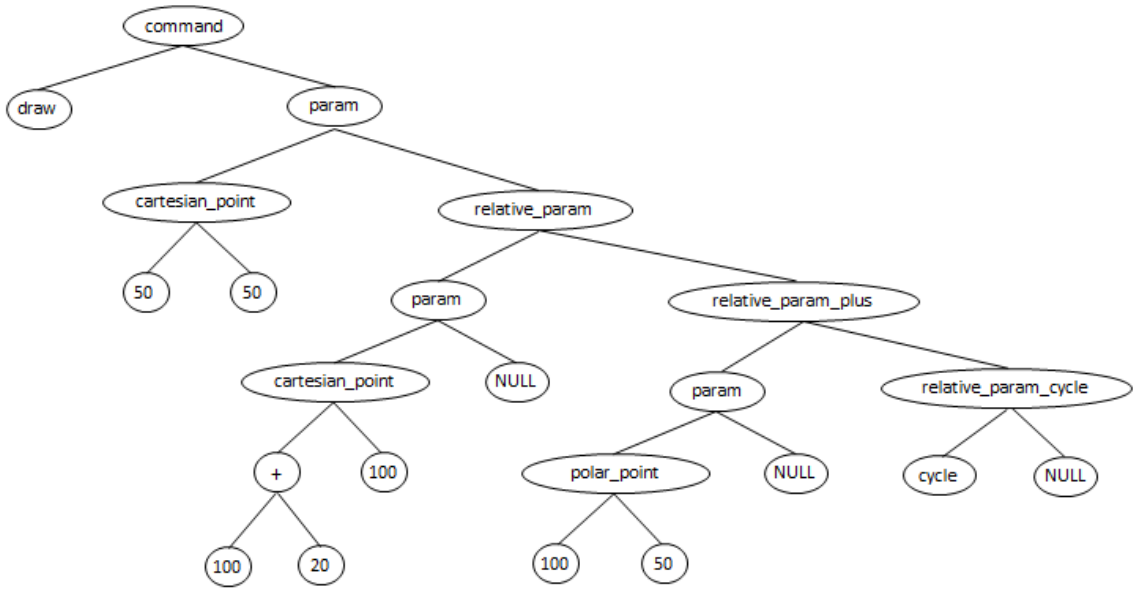
1.2.6 Les commentaires

@< $commentaire$ >@ fonctionnant de la même façon que les $/*\,*/$ en C/C++.

2 L'arbre syntaxique

Au cours de l'analyse syntaxique, nous formons l'arbre abstrait syntaxique contenant l'ensemble des instructions du programme à compiler. Celui-ci se crée nœud par nœud depuis le bas, nous permettant de stocker facilement les instructions détectées par la grammaire pour ensuite les analyser durant la génération de code.

Voici par exemple l'arbre abstrait syntaxique formé après analyse de la commande suivante :
draw(50,50)--(100 + 20,100)--+(100 : 50)--cycle ;



Comme vous pouvez le voir une commande est représentée par un noeud "command" ayant comme fils gauche le nom de cette commande et comme fils droit ses paramètres.
Un point peut alors s'écrire comme un "cartesian_point" ou "polar_point" suivant son type mais il aura toujours un fils gauche contenant son membre gauche et un fils droit contenant son membre droit. Ce membre peut être un nombre ou une variable scalaire, dans ces deux cas un nœud de type "number" est créé contenant la valeur de celui-ci, ou une expression arithmétique. Dans cette situation un nœud de type "binary" est créé, contenant le symbole arithmétique reconnu et avec comme fils les membres gauches et droits de l'expression, ceux-ci pouvant de nouveau être une expression, une variable ou un entier.
Les coordonnées relatives sont stockés de façon similaire mais le nœud père du point s'appellera alors "relative_param_plus" et non pas "param" ou "relative_param" pour être différencié lors de l'écriture. Les cycles eux, se nomment "relative_param_cycle" et ne possèdent aucun fils.

Nous avons donc ci-dessus un arbre pouvant contenir des commandes relativement simples qui s'enchaîneront grâce à des nœuds "list_of_instructions" pouvant avoir comme fils des commandes ou d'autres listes d'instructions.

3 Génération du code

Pour générer le code nous stockons toutes les commandes de l'arbre dans une file que nous parcourons ensuite à trois reprises, deux pour les pré-traitements puis une dernière pour l'écriture.
Nous allons ici vous parler de la génération de code en fonction de l'arbre et non pas du début de fichier contenant les librairies et les initialisations de contextes, ni de la fin contenant leurs destructions, qui sont commune à chaque exécution du programme.

3.1 Pré-traitements

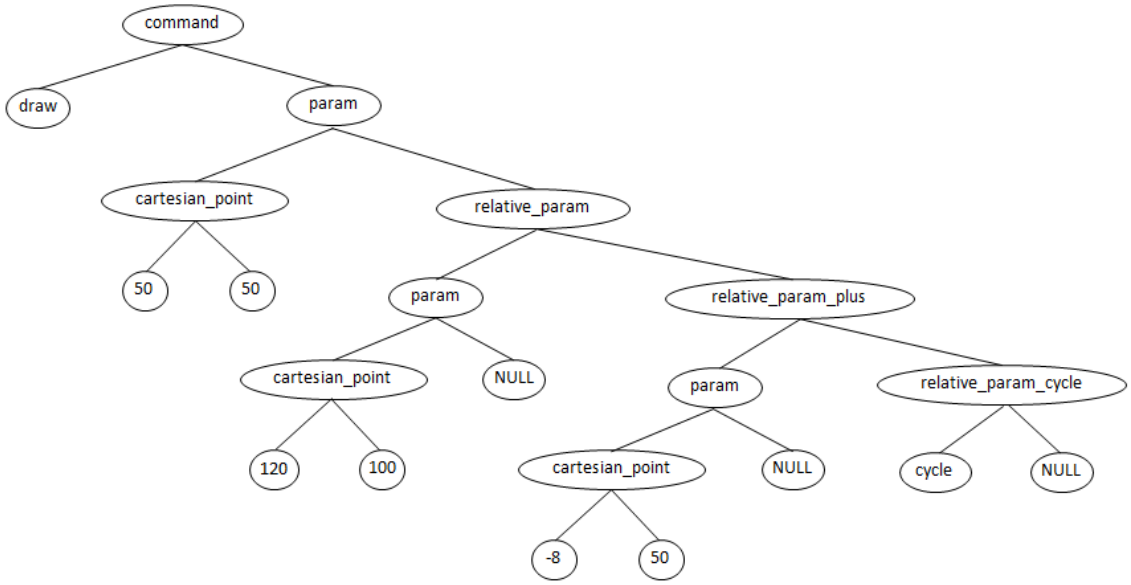
3.1.1 Premier passage

Durant le premier passage, l'arbre syntaxique est simplifié. Les points en coordonnées polaires sont convertis en points cartésiens grâce aux opérations suivantes :

- $abscisse = rayon * cos(angle * radian)$
- $ordonne = rayon * sin(angle * radian)$

et les expressions arithmétiques sont calculées récursivement.

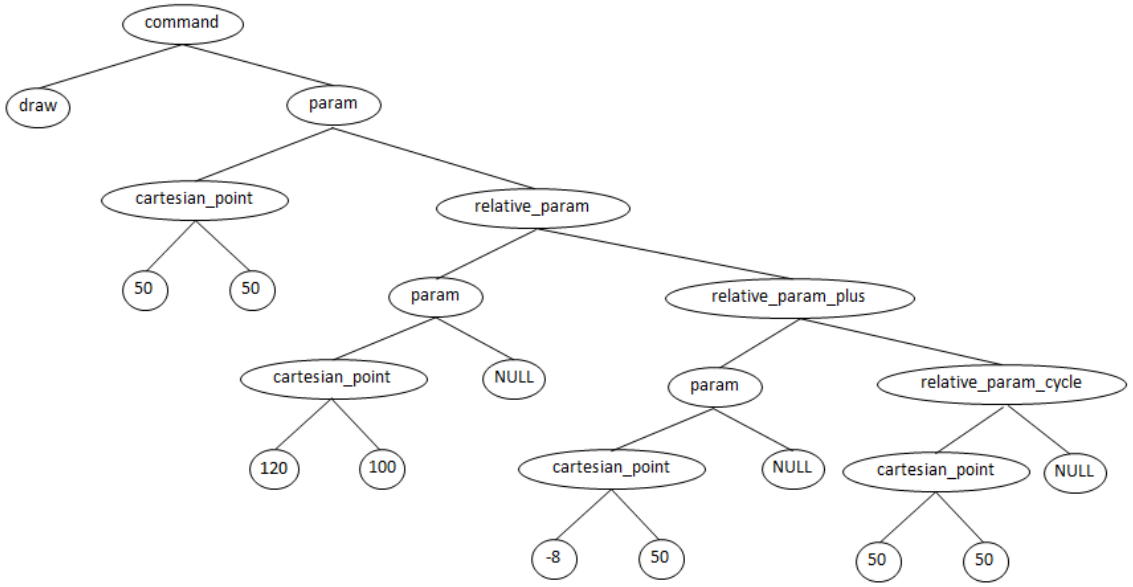
Voici par exemple l'arbre syntaxique, formé après analyse et premier passage, de la commande suivante :
draw(50,50)--(100 + 20,100)--+(100 : 50)--cycle ;



3.1.2 Second passage

Durant le second passage, on résout les cycles en remplaçant ceux-ci par le premier paramètre de la commande. Cette tache est effectuée séparément du premier passage pour être sûr que le cycle soit remplacé par des coordonnés cartésiens exploitables directement.

Voici par exemple l'arbre syntaxique, formé après analyse et pré-traitement complet, de la commande suivante :
draw(50,50)--(100 + 20,100)--+(100 : 50)--cycle ;



3.2 Génération du code

Après le pré-traitement, l'arbre abstrait syntaxique contient des informations qui ne demandent qu'à être réécrite.

Nous utilisons la file pour obtenir une commande à la fois. Nous avons le nom de la commande dans le fils gauche permettant de savoir qu'elle fonction réécrire ensuite mais également les paramètre possibles dans le fils droit que nous analysons récursivement.
Le premier point d'une commande génère obligatoirement un *cairo_move_to* mais pas les suivants. Si nous avons un cycle, celui-ci a maintenant comme fils un point donc nous pouvons également écrire un *cairo_line_to*, de même que si c'était un autre point, qui va créer une ligne entre le point précédent et celui-ci. Mais si nous avons un "relative_param_plus" nous écrivons $cairo_relative_line_to$ qui crée également une ligne entre les deux points mais en prenant les coordonnées du second point relativement au premier. Une fois que tout les paramètres d'une commande ont été parcourus, nous pouvons écrire un *cairo_stroke* si nous avons une commande "draw" ou un *cairo_fill* dans l'autre cas.

Ainsi la commande $draw(50,50) - -(100+20,100) - - + (100:50) - - cycle$; génère le code suivant :

```
cairo_move_to(cr , 50 , 50);
cairo_line_to(cr , 120 , 100);
cairo_rel_line_to(cr , -8 , 50);
cairo_line_to(cr , 50 , 50);
cairo_stroke(cr);
```

4 Les variables

4.1 Détection des variables

Nous partons du principe que tout mot non déjà détecté par le lexer est une variable.

Chaque variable est stocké dans une structure symtab fonctionnant comme une pile, qui contient le nom de la variable, son type c'est à dire soit scal pour un entier soit path pour un point ou un chemin soit pict pour une image, sa valeur c'est-à-dire soit un entier soit un arbre correspondant à un chemin ou à une image, ainsi que la variable suivante.
le lexer teste la présence de la variable dans symtab, si celle-ci n'y est pas présente c'est qu'elle n'a pas était déclaré donc le lexer retourne le token ID qui n'est accepté dans la grammaire que pour une déclaration. Par contre si celle-ci y est présente alors on teste son type et le lexer retourne le token correspondant (par exemple $PATH\_ID$ si elle est de type path).
Grâce à ce système on ne peut pas effectuer d'affectation sur une variable non déclarée.

Actuellement les déclarations doivent s'effectuer séparément des affectations pour une question de temps, les variables étant pleinement opérationnelles comme ceci, nous n'avons pas trouvés la possibilité de faire les deux à la fois très importante, préférant nous attarder sur d'autres aspects.

4.2 Gestion des contextes

Avec l'implémentation ci-dessus, se posait le problème des contextes car lors de l'entrée dans un nouveau contexte la variable se trouvait écraser si on la modifiait et on ne pouvait donc pas récupérer son ancienne valeur lors de la sortie de ce contexte.
La solution retenue a été de créer une structure scope fonctionnant encore comme une pile, contenant une symtabcell représentant le contexte courant. Cette symtabcell contient la symtab courante mais également une autre symtabcell représentant le contexte antérieur.
Ainsi lors de l'entrée dans un nouveau contexte on crée une nouvelle symtabcell, avec une symtab copie de la précédente, dans laquelle se feront les accès aux variables, que l'on supprimera lors de la sortie de ce contexte pour récupérer l'ancienne symtabcell restée inchangée.

5 Les images

Une image consiste en un bloc *image* contenant une liste d'instructions. Les listes d'instructions sont déjà codées donc on peut juste créer un noeud "image" ayant comme fils l'arbre représentant la liste d'instructions pour avoir un arbre correct.
Une image est dessinée en faisant appel à "draw" et on peut vouloir dessiner plusieurs images à la suite alors que mélanger des images avec des chemins classiques doit être interdit. On ajoute donc des nœuds "list_of_pict" ayant comme fils une image ou une autre liste d'images pour permettre le premier point sans permettre le second car les images ne sont pas accessibles depuis un nœud "param".

6 Rotate et translate

Ces deux fonctions fonctionnent sur le même principe que nos fonctions de génération de code sauf qu'aux lieux d'écrire des fonctions elles renvoient un arbre. Avant tout on clone l'arbre à transformer puis on applique les premier et second passages décrit plus haut pour avoir des points sur lesquels effectuer nos calculs. Ensuite nous regardons si nous avons juste un chemin sur lequel appeler la fonction auxiliaire correspondante ou une image, dans ce second cas on enfile la liste des commandes comme pour la génération de code et on appellera la fonction auxiliaire pour chacun des chemins.
Ces fonctions auxiliaires vont alors effectuer récursivement les calculs nécessaires à la translation ou à la rotation sur chaque point du chemin.

Pour une translation de vecteur(x, y) à un point (a, b) on effectue les opérations suivantes :
- $a+ = x$
- $b+ = y$

tandis que pour une rotation de centre(x,y), d'angle o sur un point (a, b) on effectue :
- $a = cos(o) * (a - x) - sin(o) * (b - y) + x$
- $b = sin(o) * (a - x) + cos(o) * (b - y) + y$

La fonction retourne alors le clone sur lequel a été effectué toutes les opérations.

7 Les conditionnelles

Pour appliquer une instruction if/else, nous avons, avant tout, dû implémenter les comparaisons sur le même modèle que les expressions arithmétiques.
Ensuite lorsque une conditionnelle est détectée par la grammaire nous évaluons directement le résultat de la condition et n'écrivons que la liste d'instructions correspondant au if si celle-ci est vrai ou au else sinon (s'il n'y a pas de else, on met un NULL). Le code généré ne contient alors que la branche choisi.

8 Personnalisation des traits

Ceci était une question bonus que nous avons partiellement effectuée.

Nous gérons line_width et line_color comme les commandes précédentes et lors de la génération de code nous écrivons respectivement soit *cairo_set_line_witdh* soit *cairo_set_source_rgb*.
Pour cette deuxième fonction les paramètres doivent être trois float compris entre 0 et 1 or nous avons implémentés les nombres comme des entiers. Pour éviter de changer cette implémentation tardivement nous prenons des int en paramètre mais que nous considérons entre 0 et 100 puis que nous divisons par 100.0 pour obtenir des paramètres corrects, cairo ramenant à 1 toute valeur supérieur.
Nous ne nous occupons pas des styles de trait mais il faudrait autoriser par le lexer les différents styles possibles que nous pourrions passer en paramètre à la nouvelle fonction et stocker ceux-ci dans le nom du nœud correspondant à son paramètre pour ensuite pouvoir appeler la fonction de cairo correspondante durant la génération de code.

9 Conclusion

A la fin nous avons donc un langage de dessin opérationnel comprenant des variables, des images, des conditionnelles, une personnalisation des traits basiques, ainsi que des expressions arithmétiques et différents types de points.
L'implémentation des fonctions est quelque chose de manquant, même si leur principe et fonctionnement sont théoriquement proche de celui des images nous avons passé trop de temps sur les questions précédentes pour nous pencher dessus.

Le projet nous a permis de mettre en œuvre ce que nous avons appris lors des amphis et les séances de TD concernant bison et flex mais également de coupler ces derniers avec du code C plus conséquent et de mieux appréhender les possibilités et les difficultés lors de la création d'un nouveau langage mais également de mieux comprendre les langages que nous utilisons couramment.