Projet d'analyse syntaxique

Université Bordeaux 1 — 2012–2013

1 Modalités de réalisation

Deux sujets sont proposés au choix à réaliser en C/flex/bison par groupes 3 personnes maximum. Chaque membre d'un groupe doit réaliser une partie significative du travail de programmation. L'écriture seule du rapport, par exemple, est considérée comme insuffisante. Chaque membre d'un groupe doit connaître les options générales choisies par le groupe ainsi que les difficultés rencontrées. Chacun devra exposer individuellement sa contribution lors de la soutenance (**13 et 14 mai 2013**).

Le projet sera présenté dans un rapport synthétique (10 pages typiquement) illustré par des exemples, exposant les problèmes rencontrés et les choix effectués pour les résoudre. Remise du source et du rapport par mail à aspp3@labri.fr : **vendredi 10 mai 2013 à midi** au plus tard. Le fichier transmis par mail doit être un fichier de type `.tar.gz` contenant un seul répertoire à vos noms, dans lequel se trouvera :

- le fichier du rapport sous forme pdf : `rapport.pdf`.
- un répertoire `./src/` contenant les sources du projet.

La notation tiendra compte du degré de réalisation du sujet, de la qualité du code (portabilité, lisibilité, documentation), de la qualité des tests traités avec succès et présentés en soutenance, de la qualité du rapport, et de la présentation orale. Attention : la notation pourra varier à l'intérieur d'un groupe en cas de travail trop inégal, et tout plagiat détecté sera sévèrement sanctionné.

En cas de difficultés ou de questions, n'hésitez pas à contacter l'équipe pédagogique aspp3@labri.fr.

Sujet 1 : langage de programmation de dessins vectoriels

Ce projet propose d'élaborer et d'implémenter un langage de programmation de dessin. L'implémentation en `C` de la partie graphique se basera sur la bibliothèque `cairo`, qui permet de dessiner sur une surface qui peut tout aussi bien être un fichier `.pdf`, `.png`, `.ps`, *etc.*, ou une fenêtre `GTK`, `Quartz`, *etc.* Le code suivant, disponible sur la page du cours permet de dessiner la figure 1. Il faut noter que l'axe des y croît vers le bas et que l'axe des x croît vers la droite.

FIGURE 1 – Dessin généré par le programme `ex1.c`

```
#include <cairo.h>
#include <cairo-pdf.h>

int main(void){
  cairo_surface_t *surface;
```

```
  cairo_t *cr;

  //Creation de la surface pdf associee au fichier ex1.pdf
  cairo_surface_t* pdf_surface =
    cairo_pdf_surface_create("ex1.pdf",50,50);

  //Creation du contexte cairo associe a la surface
  cr=cairo_create(pdf_surface);
  //Place le point courant en (10,10)
  cairo_move_to(cr,10,10);
  //Enregistrer la ligne du point (10,10) au point (50,50)
  cairo_line_to(cr,50,50);
  //Met la largeur de trait a 10
  cairo_set_line_width (cr, 10.0);
  //Tracer la ligne
  cairo_stroke(cr);
  //Liberation du contexte
  cairo_destroy(cr);
  //Liberation de la surface
  cairo_surface_destroy(pdf_surface);
  return 0;
}
```

Le site web consacré à cette bibliothèque a pour adresse : http://www.cairographics.org/. Sa documentation se trouve à l'adresse : http://www.cairographics.org/manual/index.html.
Remarque : pour compiler ce programme, il faut utiliser la commande :

```
gcc -o ex1.out `pkg-config -cflags -libs cairo` ex1.c
```

Une fois ce programme compilé, son exécution produit un fichier `pdf` contenant l'image de la figure 1.

Objectifs

L'objectif du projet est de modéliser et d'implémenter avec `flex`, `bison` et `C` un langage de description de dessins vectoriels dont la réalisation sera assurée par `cairo`. Le projet va être structuré par une série d'objectifs :

1. Nous allons commencer par des commandes permettant de construire des lignes brisées de la forme : `draw p1 -- p2 -- ... -- pn;` ou `fill p1 -- p2 -- ... -- pn;`. Les points peuvent être représentés de deux façons, en coordonnées cartésiennes `(x,y)` ou en coordonnées polaires `(angle:rayon)`. On autorise l'utilisation d'expressions pour calculer les coordonnées. On pourra par exemple écrire une commande :

   ```
   draw (0,10)--(10,10)--(10,0)--((10+10)/3,(10+10)/3);
   ```

   Écrivez les fichiers `bison` et `flex` qui permettent d'analyser et d'exécuter avec `cairo` une liste de telles commandes.
2. On souhaite enrichir la syntaxe des chemins en permettant l'utilisation de coordonnées relatives et de construire des chemins fermés. Pour cela, on pourra utiliser des points de la forme `+pt` pour indiquer que le point considéré dans le chemin est le point précédent translaté de `pt`. On permettra également d'utiliser des points de la forme `cycle` qui désigneront le premier point du chemin que l'on construit. Par exemple la commande :

```
draw (0,5)--+(0,1)--cycle--(4,0)--cycle--+(-5,1);
```

devra être équivalente à la commande :

```
draw (0,5)--(0,6)--(0,5)--(4,0)--(0,5)--(-5,6);
```

3. On souhaite maintenant étendre notre langage avec des variables. Ces variables pourront contenir ou bien des scalaires, des points, ou encore des chemins.
4. Une image est une liste de commandes qui ne sont pas exécutées immédiatement. Elles ont une syntaxe délimitée par bloc `image{... }`, avec les contraintes suivantes :
   (a) les variables déclarées dans un tel bloc seront locales à ce bloc,
   (b) les blocs peuvent être imbriqués.

   Une image peut être stockée dans une variable et peut être dessinée avec la commande `draw`.
5. Afin d'exploiter au mieux les variables, on souhaite ajouter la possibilité des transformer les objets graphiques avec des transformations géométriques. Aussi, on autorise des expressions de la forme `rotate(p,c,angle)` ou `translate(p,v)` où `p` peut-être un point, un chemin ou une image, `c` est le centre de rotation, `angle` est un l'angle de rotation et `v` est le vecteur de translation. Ces fonctions doivent renvoyer l'objet transformé du bon type.
6. Ajouter des conditionnelles avec la syntaxe de `C`, en ajoutant bien entendu des comparateurs.
7. Ajouter des boucles `for` avec la syntaxe de `C`.

Partie optionnelle

8. Ajouter des déclarations de fonctions.
9. Ajouter les fonctions (avec possibilité d'appels récursifs).
10. Ajouter des fonctionnalités permettant de gérer les styles de dessins, couleur, épaisseur de trait.
11. Ajouter des moyens de composition d'image gérant automatiquement les alignements.

Il convient que, dès le départ, vous utilisiez des messages d'erreurs qui vous permettent de localiser les erreurs dans le code analysé. Nous vous conseillons d'utiliser l'analyse syntaxique pour générer un arbre de syntaxe abstraite qui servira à exécuter le code dans `cairo`. Cela vous permettra d'isoler les aspects syntaxiques des aspects sémantiques et vous permettra d'éviter des erreurs. Aussi, il conviendra à chaque évolution du projet de faire évoluer cet arbre abstrait, son interprétation et son lien avec la syntaxe.

Les questions à partir de la question 7 sont optionnelles, elles pourront être résolues dans un ordre arbitraire. Vous avez également toute latitude pour inventer et implémenter les fonctionnalités que vous souhaitez.

Sujet 2 : Buffalo, un analyseur syntaxique S/LR(1)

Ce projet demande la réalisation d'un générateur d'analyseur syntaxique. Le générateur lit une spécification de grammaire dans le fichier qui lui est donné en argument et génère un analyseur SLR(1) ou LR(1) pour cette grammaire. La spécification de la forme du fichier décrivant la grammaire est la même que pour `bison`. Les trois sections devront être présentes. Dans la première partie, on permettra en particulier le bloc littéral que le générateur se contentera de copier dans le code de l'analyseur. Les directives `%union`, `%token`, `%left`, `%right`, `%nonassoc`, `%type` devront pouvoir être traitées par le générateur.

L'analyseur doit en plus pouvoir gérer les options suivantes non standard :

– `%option dot` qui demande à l'analyseur de générer un arbre syntaxique lorsque l'entrée est syntaxiquement correcte, au format `dot` (http://www.graphviz.org/content/dot-language).

– `%option lr` qui demande à l'analyseur d'utiliser l'algorithme LR(1) au lieu de l'algorithme SLR(1). Si cette option n'est pas présente, l'algorithme SLR(1) sera employé par défaut.

Le fichier devra être compatible avec le format `bison`. En particulier, la forme d'entrée de chaque règle de grammaire sera du même type que celle employée par `bison` :

```
non-terminal:   membre droit de la règle 1 { action 1 }
              | membre droit de la règle 2 { action 2 }
              ...
              ;
```

chaque règle étant alors considérée comme distincte. Un membre droit peut par ailleurs être vide.

Le générateur d'analyseur syntaxique doit fournir :

1. l'automate SLR(1) ou LR(1) sous forme texte (même type de format que celui produit par bison, ou format graphique, au choix) ;
2. les éventuels conflits ;
3. dans le cas où la grammaire est SLR(1) ou LR(1) (selon que l'option `%option lr` a été activée), le code `C` d'un analyseur syntaxique.

Cet analyseur syntaxique généré par votre programme devra pouvoir :

– lire une chaîne de terminaux par des appels à la fonction `yylex()`. Autrement dit, pour accéder au lexème suivant, la fonction générée d'analyse syntaxique appellera la fonction d'analyse lexicale `yylex()` ;

– indiquer une éventuelle erreur syntaxique ;

– en cas d'entrée syntaxiquement correcte, appliquer les actions `{action 1}`, `{action 2}`,... spécifiées par l'utilisateur lors des réductions. On ne demande pas de gérer les actions internes aux règles ;

– si l'utilisateur a utilisé l'option `%option dot` en première partie, générer un fichier au format `dot` représentant l'arbre syntaxique créé par l'analyseur.

Les lexèmes (*tokens*) seront définis par l'utilisateur comme en `bison`, et pourront aussi être donnés sous forme littérale, représentés par un caractère. Si la grammaire n'est pas SLR(1) (ou LR(1) si l'option `%option lr` a été activée), l'analyseur doit l'indiquer. Il est également demandé de pouvoir déclarer un opérateur associatif gauche ou droite, ou non associatif, et de gérer les priorités de la même façon que le fait `bison`.

Enfin, les lexèmes doivent pouvoir porter un attribut. Comme dans `bison`, on pourra indiquer au niveau de la spécification de la grammaire des calculs d'attributs à effectuer. On se restreindra à des attributs synthétisés. Dans une action, les attributs seront notés `$$`, `$1`, `$2`,... avec la même signification qu'en `bison`. Le type des attributs sera de même donné dans une directive `%union`. Dans un premier temps, vous pourrez considérer des actions simples, du type `$$=f($i,$j,...)`. L'initialisation des attributs au niveau des feuilles de l'arbre se fera via la variable `yylval`, qui devra donc être rendue accessible à la fonction d'analyse lexicale.