UNIVERSITÉ DE LORRAINE | IUT nancy Charlemagne

DUT informatique
Langage C

Projet — L-System
8 octobre 2013

Ce projet est à réaliser de préférence par binôme.

1 Énoncé général

De nombreux êtres vivants ont une structure récursive qui se comporte comme une fractale. Ce projet a pour objectif de dessiner différentes fractales à partir de grammaires connues sous le nom de « L-System ».

1 Grammaire et langage

Une *grammaire* comporte quatre éléments :
- des *variables*,
- des *symboles terminaux*,
- un point de départ, appelée *axiome*, faisant référence à une ou plusieurs variables,
- des *règles de production*.

À titre d'exemple, les règles de production suivantes génèrent des phrases françaises :

`P` → `S V C`
`S` → Jean
`S` → Marie
`V` → marche
`V` → court
`C` → dans la rue
`C` → dans le jardin

Dans cet exemple, les variables sont `P`, `S`, `V` et `C`, l'axiome est la variable `P`, et les symboles terminaux sont « Jean », « Marie », etc. À partir de `P`, nous pouvons construire la phrase « Jean marche dans la rue » ou encore « Marie court dans la rue ». On appelle *langage* engendré par une grammaire l'ensemble des phrases [1] qu'on obtient à partir de l'axiome.

La syntaxe des langages de programmation comme le langage C est également définie par une grammaire… Cet aspect est traditionnellement traité dans les cours de compilation.

2 Grammaire récursives, interprétation graphique

a Grammaires récursives

Certaines règles de productions peuvent être *récursives*, c'est-à-dire intégrer dans leur partie droite une ou plusieurs références aux variables définissant ces règles. Ainsi, il est possible d'avoir des règles du type :

[1] Terminologiquement, on parle plutôt de *mots* dans ce contexte.

`A` → `A + B`

Si `A` est également axiome, le langage engendré contient :
- initialement : `A`
- au 1er niveau de développement : `A + B`
- au 2e niveau de développement : `A + B + B`
- au 3e niveau de développement : `A + B + B + B`
- …

Ces niveaux de développement sont appelés *générations* dans le contexte particulier des L-System, destinés initialement à la modélisation du développement d'organismes vivants.

b Interprétation graphique

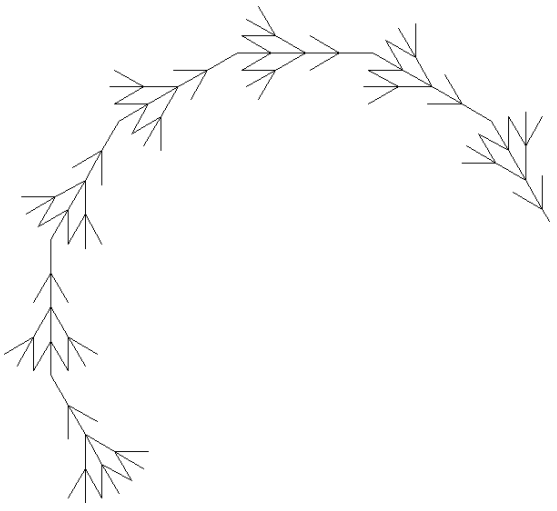
Afin de visualiser le développement d'un axiome, on peut convenir d'un certain nombre de `commandes` permettant de piloter une tortue graphique[2] . Ces commandes peuvent être des variables ou des symboles terminaux de la grammaire étudiée. Une tortue graphique est définie par une position, une direction et est capable de tracer des traits lors de ses déplacements.

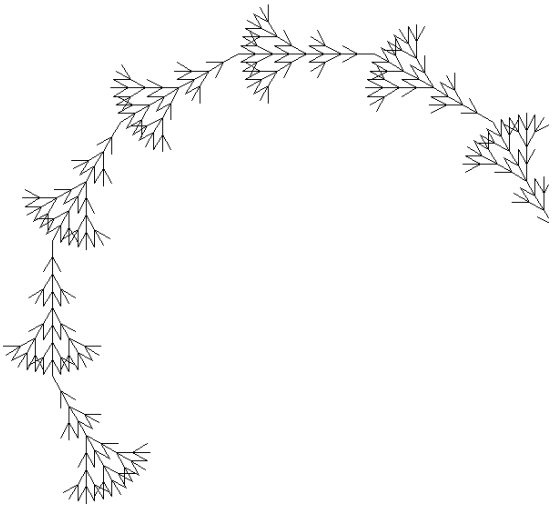
Dans le cadre de ce projet, on définit conventionnellement les commandes suivantes, permettant de piloter la tortue graphique :
- `F` : avance d'un pas en dessinant un trait
- `f` : avance d'un pas sans rien dessiner
- `+` : tourne à gauche
- `-` : tourne à droite
- `[` : sauve l'état de la tortue sur la pile
- `]` : restaure l'état de la tortue à partir de la pile

3 L-System

Les L-System[3] reposent sur les notions introduites précédemment : grammaires récursives, tortue graphique… À titre d'exemple, voici la description du L-System « chardon » :
- axiome : `F+F+F+F+F+F+F`
- unique règle de production : `F` → `F[-F][+F]F`
- nombre d'orientations : 12[4]

Voici, ci-dessous, les dessins obtenus après calcul de la 2e et de la 3e génération.





[2] Comme en LOGO, voir `http://fr.wikipedia.org/wiki/Logo_(langage)`.
[3] `http://fr.wikipedia.org/wiki/L-System`
[4] Dans ces conditions, tourner à gauche ou à droite correspond donc à un changement d'orientation de $\frac{360}{12} = 30^{\circ}$.

2 Réalisation

La réalisation de ce projet devra s'articuler autour de 6 modules :
- un dédié à la gestion d'une grammaire,
- un dédié à la gestion d'une pile (permettant d'empiler des tortues),
- un dédié à la gestion de la tortue graphique,
- un dédié à la gestion de l'interpréteur d'une grammaire,
- un module d'outils utiles,
- un module principal.

Vous aurez les modules de grammaire, d'outils, de pile, et une partie de l'interpréteur à réaliser. Les autres parties sont accessibles depuis l'ENT.

a Gestion d'une grammaire

Les grammaires correspondant aux L-System à traiter sont présentes dans des fichiers textes, tel que celui-ci :

```
chardons
12
4
F+F+F+F+F+F+F
F->F[-F][+F]F
```

Les informations présentes dans de tels fichiers sont toujours les mêmes :
- En ligne nº 1 : le nom de la grammaire.
- En ligne nº 2 : le nombre de directions à gérer pour cette grammaire. Une direction fait un angle de $\frac{360^\circ}{\text{nbredirections}}$ avec la direction précédente et la direction suivante, soit 30° sur cet exemple.
- En ligne nº 3 : la direction de départ pour la tortue graphique, sachant que conventionnellement les directions sont indicées à partir de 0 et que la direction 0 correspond à l'angle 0°.
- En ligne nº 4 : l'axiome, faisant référence à une ou plusieurs des règles présentes après.
- À partir de la ligne nº 5 : les règles de la grammaire, sachant que :
  - chaque règle tient sur une unique ligne,
  - chaque règle est identifiée (nommée) par une lettre alphabétique majuscule,
  - un nom de règle est suivi immédiatement (sans espace) par une flèche (`->`),
  - une flèche est suivi immédiatement (sans espace) par son développement, également appelé sa partie droite.

On précise par ailleurs que :
- À l'intérieur de tels fichiers, **aucune** ligne ne peut excéder 79 caractères.
- Il ne peut pas y avoir plus de 10 règles à l'intérieur de tels fichiers.
- Il sera utile de mémoriser (c'est-à-dire de stocker dans la structure permettant de représenter une grammaire) la longueur de chaque partie droite des différentes règles. Ces informations seront utilisées par la suite…

À partir des indications données, créer un module `Grammaire`, contenant une structure `Grammaire` permettant de stocker toutes les informations relatives à une grammaire et disposant des fonctions suivantes :

```
void grammaire_lire(char *nomfic, Grammaire *g);
void grammaire_afficher(Grammaire g);
int grammaire_trouver(Grammaire *g, char cmd);
```

Explications :
- la première fonction doit permettre de remplir une structure `Grammaire` à partir d'un fichier (dont le nom est fourni) en contenant une,
- la seconde fonction doit permettre d'afficher une grammaire (afin de vérifier les informations lues),

– la troisième fonction doit renvoyer l'indice (par rapport à votre représentation interne, on se doute que vous allez faire intervenir des tableaux) d'une règle à partir d'une grammaire et d'un nom de règle donnés.

Indication : il est recommandé de faire usage de **mémoire statique** pour les données de ce module. Les informations fournies dans l'énoncé doivent vous permettre de déterminer au mieux les besoins.

b Gestion d'une pile

On rappelle que les *piles* sont des listes particulières, pour lesquelles deux opérations principales sont autorisées :
– l'*empilage*, correspondant à une insertion en tête au niveau de la structure de liste sous-jacente,
– le *dépilage*, correspondant à une suppression en tête au niveau de la structure de liste sous-jacente.

À partir des indications données, créer un module `Pile`. Le fichier d'en-tête de ce module vous est fourni :

```
#ifndef PILE_H_INCLUS
#define PILE_H_INCLUS

#include "tortue.h"

typedef struct pile_cell {
  float                 x;       /* Position en X d'une tortue  */
  float                 y;       /* Position en Y d'une tortue  */
  int                   dir;     /* Direction d'une tortue      */
  struct pile_cell  *suc;        /* Chaînage                    */
} Pile_Cellule;

typedef Pile_Cellule *Pile;

/* Retourne une pile vide */
Pile pile_initialiser();

/* Retourne vrai si la pile est vide */
int pile_vide(Pile p);

/* Empile les informations utiles de t sur p et retourne la nouvelle pile */
Pile pile_empiler(Pile p, Tortue *t);

/* Dépile un élement de p pour remplir t et retourne la nouvelle pile p */
Pile pile_depiler(Pile p, Tortue *t);

/* Affiche le contenu de la pile */
void pile_afficher(Pile p);

/* Libère une pile en la vidant complètement */
void pile_liberer(Pile p);

#endif
```

c Module d'outils

Ce module d'outils sera doté d'une seule fonction, permettant de réserver dynamiquement de la place, avec les vérifications nécessaires, pour une nouvelle chaîne dont la longueur (en caractères) est fourni en paramètre. L'en-tête de cette fonction vous est fourni :

```c
char *tools_nouvelle_chaine(int length);
```

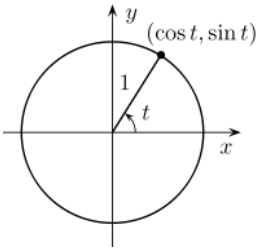
d Gestion de l'interpréteur

L'interpréteur est le module qui, à partir d'une grammaire, va générer les différents niveaux de développement requis et piloter la tortue graphique pour permettre une représentation graphique du résultat.

Avant de parvenir à la représentation graphique, plusieurs étapes sont nécessaires :

1. **Production de la génération demandée d'une grammaire**, ce qui permet de produire un *mot résultat* (voir `inter_generer()` en particulier).
2. **Nettoyage du *mot résultat*** afin d'en supprimer tous les caractères qui ne correspondent pas à des commandes graphiques (voir `inter_nettoyer()` en particulier).
3. **Simulation de déplacement de la tortue à partir du mot final nettoyé** (voir `inter_calc_dimension()` en particulier). Cette étape est nécessaire car on ne peut pas prédire à l'avance quelles dimensions aura le dessin produit par la tortue. On va donc, dans un premier temps, déplacer la tortue comme si on produisait le dessin (ce qui ne sera pas encore le cas) en prenant soin, à chaque déplacement de la tortue, de prendre en considération ses nouvelles coordonnées pour mettre à jour les attributs de l'interpréteur permettant de mémoriser l'abscisse et l'ordonnée minimales/maximales.

   Lors de cette étape, les déplacements de la tortue sont d'une longueur d'une unité à chaque fois ; la nouvelle position de la tortue se calcule donc à partir de la position précédente grâce aux valeurs précalculées des sinus et cosinus stockées dans la structure de la tortue (la figure ci-dessous en rappelle le principe).



   À la fin de la simulation, l'abscisse et l'ordonnée minimales/maximales déterminent ainsi la *fenêtre* de dessin, *i.e.* la portion rectangulaire de l'espace qui est utilisée pour le dessin. Un simple calcul de ratio [5] et de décalage en X et en Y permettront alors de faire correspondre plus tard, lors du dessin *effectif*, le déplacement de la tortue à la taille d'image résultat souhaitée (ce paramètre est fourni en ligne de commande au programme, voir le source du programme principal).
4. **Déplacement effectif de la tortue à partir du mot final nettoyé** (voir `inter_interpreter()` en particulier). Le ratio étant calculé (il faut considérer la plus grande des dimensions pour cela, ce qui est effectué dans le programme principal) et le décalage étant connu (il suffit de retrancher les coordonnées minimales constatées à toute nouvelle coordonnée calculée), le dessin peut être réalisé. Cette fonction vous est fournie, vous pouvez naturellement l'analyser pour en comprendre le fonctionnement.

L'en-tête de ce module d'interprétation vous est fourni :

```c
#ifndef INTERPRETE_H_INCLUS
#define INTERPRETE_H_INCLUS

#include <limits.h>

#include "tortue.h"
#include "pile.h"
#include "gramm.h"
```

[5] Facteur d'homothétie.

```c

#define INFINI INT_MAX

typedef struct
 {
   char               *mot;   /* Mot de la grammaire à interpréter graphiquement */
   int                taille; /* Taille = strlen(mot) */
   float              xmin;   /* Taille de la fenêtre de la tortue */
   float              ymin;
   float              xmax;
   float              ymax;
 }  Interprete;

void inter_init(Interprete *inter);
void inter_interpreter(Interprete *inter, float factor, Tortue *tortue, char *nom_fichier);
int inter_generer(int niveau_max, Grammaire *g, Interprete *t);
void inter_nettoyer(Interprete *inter);
void inter_calc_dim(Interprete *inter, Tortue *tortue);
Pile inter_transition(Pile p, char cmd, Tortue *tortue);
void inter_liberer(Interprete *inter);

#endif
```

Commentaires et précisions :

- Au niveau de la structure :
  - le champ `mot` pointe vers le mot courant. C'est initialement un pointeur NULL. À chaque nouvelle génération, cet attribut pointe vers le mot courant,
  - le champ `taille` contient la taille du mot courant (utilisée plusieurs fois, il est utile de la stocker),
  - les autres champs permettent de calculer la boîte englobante (*bounding-box*) du dessin généré.
- La fonction `inter_init()` a pour but d'initialiser l'interprète. Chaque champ de la structure correspondante est concerné…
- La fonction `inter_interpreter()` : permet de générer le dessin. Le corps de cette fonction vous est fourni.
- La fonction `inter_generer()` : permet de générer le mot correspondant à une grammaire et à un niveau de développement (une génération). En interne :
  - l'axiome de la grammaire est utilisé comme mot de départ (génération n° 0),
  - une boucle permet d'itérer sur les différentes générations à produire pour arriver à la génération finale,
  - la structure de l'interprète est mise à jour à la fin de cette fonction.

  Cette fonction est certainement la plus délicate du projet. Attention aux considérations suivantes :
  - toutes les manipulations de chaînes doivent être **dynamiques**,
  - avant la calcul d'une nouvelle génération, il est nécessaire de calculer si un débordement mémoire est possible. Pour cela, on considère le mot actuel et on calcule la longueur, et juste la *longueur*, du futur mot. Il suffit pour cela de parcourir le mot courant et de calculer au fur et à mesure la place mémoire nécessaire pour le futur mot, en tenant compte de la longueur des différentes règles (cette information se trouve dans la structure de la grammaire). Si la valeur obtenue dépasse les possibilités de représentation de longueur d'un mot (la capacité d'un entier), on arrête le calcul de générations prématurément… La fonction renvoie un booléen, indiquant s'il y a eu détection d'un possible débordement lors du calcul de la génération demandée,
  - si un débordement n'est pas détecté, on dispose déjà de la longueur du futur mot. Il ne reste alors plus qu'à allouer de l'espace mémoire pour ce mot et à le générer, en parcourant à nouveau le mot courant.
- La fonction `inter_nettoyer()` permet de supprimer du mot courant tous les caractères qui ne sont pas interprétés graphiquement (donc tous les caractères sauf `F`, `f`, `+`, `-`, `[` et `]`).
- La fonction `inter_calc_dimension()` : interprète graphiquement le mot, sans toutefois générer de dessin, afin de mettre à jour les champs de l'interpréteur permettant de gérer la boîte englobante.
- La fonction `inter_transition()` : permet l'exécution d'une seule commande graphique, en mettant à jour la pile et la tortue utilisées.

– La fonction `inter_liberer()` permet de libérer les ressources dynamiques utilisées par la structure.

3 Comment rendre le projet ?

*Votre projet va faire l'objet d'un traitement automatisé. Merci de suivre **scrupuleusement** les consignes suivantes. Des points seront automatiquement ôtés si une ou plusieurs de ces consignes ne sont pas suivies.*

1 Consignes générales

1. Votre projet devra obligatoirement être rendu sur l'ENT. Une section « Remise de projet », présente dans le cours en ligne correspondant à ce cours, sera disponible jusqu'au lundi 4 novembre à 22h00. Si vous avez déjà déposé un projet, il vous sera possible d'en déposer une ou plusieurs versions modifiées jusqu'au délai final. **Attention**, que vous ayez ou pas déposé votre projet, il sera **impossible** de déposer un projet passé le délai final. À vous de prendre vos dispositions.
2. **Aucun fichier ou répertoire** de votre projet ne doit contenir, **dans son nom**, des caractères tels que des espaces, des caractères accentués ou des caractères de ponctuation (à l'exclusion des caractères « `_ - .` »).
3. Votre projet devra être une archive, **obligatoirement** dans l'un des formats suivants (à l'exclusion de tout autre) : `zip`, `rar`, `tar.gz`, `tgz`. Votre choix de format est désigné ci-dessous par l'`extension`.
4. Le nom de votre archive devra **obligatoirement** suivre l'une des syntaxes suivantes :
   – une seule personne : `TP_Nom.extension` (exemple : `TP_Dupont.tar.gz`)
   – deux personnes : `TP_Nom1_Nom2.extension` (exemple : `TP_Durand_Dupont.zip`)
   – trois personnes : `TP_Nom1_Nom2_Nom3.extension` (exemple : `TP_Durand_Dupont_Martin.zip`)
   Si un nom de famille est composé, ne prenez en compte que le premier mot du nom (sauf s'il s'agit d'une particule, que vous pouvez concaténer au nom — exemple : `TP_DeLaFontaine.zip`). Si un nom de famille contient une lettre accentuée, remplacez là par la lettre non accentuée correspondante.
5. Vous avez le choix, dans votre archive, de :
   – placer directement les fichiers et répertoires de votre projet à la racine,
   – placer un unique répertoire à la racine (aucun nom imposé), contenant les fichiers et répertoires de votre projet.

2 Consignes propres au projet

1. Votre archive **devra** contenir :
   – les sources `.h` et `.c`, commentés et prêts à compiler
   – un fichier Makefile, générant **obligatoirement** par défaut (c'est-à-dire lorsqu'il est appelé sans paramètre) un programme appelé `lsys` et doté d'une cible `clean` permettant de faire le ménage
   – un fichier `tests.txt`, dans lequel vous expliquerez les tests que vous avez réalisés
   – un fichier `readme.txt`, que vous pourrez utiliser pour me signaler tout point que vous souhaitez porter à ma connaissance.
2. Votre archive **pourra** contenir :
   – le ou les fichiers personnels utilisés pour vos tests
3. Votre archive **ne devra pas** contenir :
   – les fichiers résiduels de compilation (`.o` par exemple)
   – les fichiers de sauvegarde (les fichiers `~`, les fichiers `.bak`)

Vous avez fini votre projet ? Vous vous apprêtez à le rendre ? Merci de relire l'énoncé et de vérifier que toutes les contraintes qui vous ont été données ont bien été respectées. Chaque contrainte est associé à un nombre de points dans la notation...