Programmation Fonctionnelle
Projet – Les L-Systèmes

octobre 2020

Un système de Lindenmayer ou *L-système* est un système de réécriture qui permet de générer des graphiques modélisant la croissance des plantes. Le but de ce projet est d'écrire un programme Caml permettant de définir des L-systèmes et de visualiser les graphiques qu'ils génèrent.

1 Les L-systèmes

Les définitions ci-dessous sont purement formelles, et ne prendront véritablement sens qu'à la section suivante. Un système de Lindenmayer n'est rien de plus qu'un mécanisme de production de suites de symboles : une suite initiale, une règle permettant de transformer une suite produite en une nouvelle suite par substitution de ses symboles.

1. Étant donné un ensemble de symboles $\mathcal{S}$, une *chaîne parenthésée* sur $\mathcal{S}$ est une chaîne d'éléments de $\mathcal{S}$ dont certaines sous-chaînes sont encadrées par des crochets « [» et «] ». Par exemple,

   $$\mathrm{F[PF][F[PF]MF]MFF]}$$

   est une chaîne parenthésée sur l'ensemble $\mathcal{S} = \{\mathrm{F}, \mathrm{P}, \mathrm{M}\}$.

2. Une *substitution* sur un ensemble de symboles $\mathcal{S}$ est une fonction associant à des éléments de $\mathcal{S}$ des chaînes parenthésées sur $\mathcal{S}$. Par exemple, si $\mathcal{S} = \{\mathrm{F}, \mathrm{P}, \mathrm{M}\}$ la fonction telle que

   $$\begin{cases} \mathrm{F} & \mapsto & \mathrm{F[FM]} \\ \mathrm{M} & \mapsto & \mathrm{[FP]} \end{cases}$$

   est une substitution sur $\mathcal{S}$.

3. *Appliquer* une substitution à une chaîne parenthésée consiste à parcourir cette chaîne en remplaçant chaque symbole dans le domaine de cette substitution par son image. Par exemple, en appliquant la substitution précédente à FPM, on obient la chaîne F[FM]P[FP].

4. Un *L-système* est la donnée d'un triplet $\mathcal{L} = (\mathcal{S}, w, \sigma)$ où :

- $\mathcal{S}$ est un ensemble (fini) de symboles,
- w est une chaîne parenthésée sur $\mathcal{S}$,
- σ est une substitution sur $\mathcal{S}$.

5. Les chaînes produites par les *itérations* de $\mathcal{L} = (\mathcal{S}, w, \sigma)$ sont définies par :
   - la chaîne produite par 0 itérations est w,
   - si v est la chaîne produite par n itérations, la chaîne produite par $n+1$ itérations est celle obtenue en appliquant σ à v.

2 Interprétation graphique des L-systèmes

Les symboles d'une chaîne parenthésée peuvent être interprétés comme des opérations graphiques, produisant souvent – mais pas toujours – un tracé de forme arborescente, rappelant la structure d'une plante. La suite de chaînes produite par les itérations d'un L-système rappelle parfois quant à elle la croissance naturelle de ces plantes – l'intention de Lindenmayer était précisément de modéliser ce phénomène.

2.1 La Tortue

Les opérations graphiques résultant de l'interprétation d'un L-système seront vues ici comme effectuées par une *tortue*, un robot virtuel muni d'un crayon et se déplaçant sur un plan. La Tortue peut effectuer trois sortes d'actions :

- pivoter sur place d'un certain angle négatif ou positif,
- avancer d'une certaine longueur, crayon levé,
- avancer d'une certaine longueur, crayon abaissé.

Plus formellement, l'état courant de la Tortue est un triplet (x, y, θ) ou (x, y) sont les coordonnées de la Tortue dans le plan et θ est l'angle formé par sa direction de déplacement et l'axe des x. Aux trois sortes d'actions effectuables par la Tortue correspondent trois sortes d'instructions. Deux instructions supplémentaires permettent de sauvegarder et restaurer son état :

- `turn a` : ajouter `a` degrés à θ,
- `move n` : avancer de `n` unités,
- `line n` : avancer de `n` unités, en traçant un segment,
- `store` : mémoriser l'état courant,
- `restore` : restaurer le dernier état mémorisé non encore restauré.

Par exemple, la suite de commandes suivante tracera un triangle équilatéral de côté 10 : `line 10`, `turn 120`, `line 10`, `turn 120`, `line 10`.

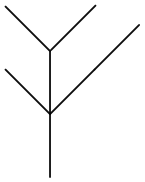


FIGURE 1 – F[PF][F[PF]MF]MFF

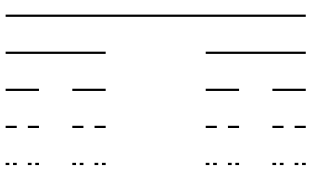


FIGURE 2 – Ensemble de Cantor

2.2 Interprétation des chaînes

Une *interprétation* d'un ensemble de symboles $\mathcal{S}$ est la donnée pour chaque symbole de $\mathcal{S}$ d'une suite d'instructions pour la Tortue. Cela peut-être, par exemple, pour $\mathcal{S} = \{\mathrm{F}, \mathrm{P}, \mathrm{M}\}$:

$$\begin{cases} \mathrm{F} & \mapsto & \texttt{line 10} \\ \mathrm{P} & \mapsto & \texttt{turn 45} \\ \mathrm{M} & \mapsto & \texttt{turn (-45)} \end{cases}$$

L'*interprétation d'une chaîne* sur $\mathcal{S}$ relativement une interprétation de $\mathcal{S}$ est définie par la suite d'instructions obtenue en parcourant cette chaîne et en transformant :

- chaque symbole de $\mathcal{S}$ en son interprétation,
- le symbole « [» par `store`,
- le symbole «] » par `restore`.

Par exemple, avec l'interprétation précédente, la chaîne F[PF][F[PF]MF]MFF produira l'arbre de la Figure 1

2.3 Evolution de l'interprétation des chaînes des L-systèmes

Exemple : L'ensemble de Cantor

$$G = (\{\mathrm{A}, \mathrm{B}\}, \mathrm{A}, (\mathrm{A} \mapsto \mathrm{ABA}, \mathrm{B} \mapsto \mathrm{BBB})).$$

Ce système produit par itération les chaînes suivantes :

0. A
1. ABA
2. ABABBBABA
3. ABABBBABABBBBBBBBBABABBBABA
4. $\cdots$

Si l'on interprète le symbole A par `line 10` et le symbole B par `move 10`, les interprétations des chaînes produites par ce L-système décrivent la construction de l'*ensemble de Cantor* (Figure 2).

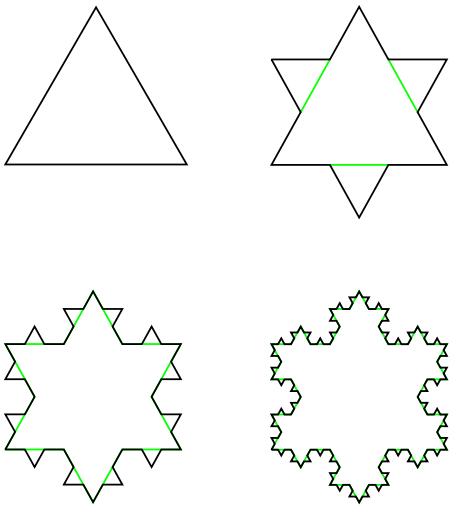


FIGURE 3 – Courbe de von Koch.

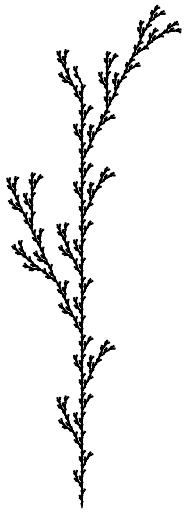


FIGURE 4 – Illustration tirée de [1].

Exemple : Courbe de von Koch

$$G = (\{\mathrm{A}, \mathrm{P}, \mathrm{M}\}, \mathrm{APPAPPA}, (\mathrm{A} \mapsto \mathrm{AMAPPAMA})).$$

Ce système produit les chaînes suivantes :

0. APPAPPA
1. AMAPPAMAPPAMAPPAMAPPAMAPPAMA
2. $\cdots$

En interprétant A comme `line 10`, P comme `turn 60` et M comme `turn -60`, les interprétations décrivent la construction de la *courbe de von Koch* (Figure 3).

Exemple : Une plante

$$G = (\{\mathrm{A}, \mathrm{P}, \mathrm{M}\}, \mathrm{A}, (\mathrm{A} \mapsto \mathrm{A[PA]A[MA]A})).$$

Ce système produit :

0. A
1. A[PA]A[MA]A
2. A[PA]A[MA]A[PA[PA]A[MA]A]A[PA]A[MA]A[MA[PA]A[MA]A]A[PA]A[MA]A
3. $\cdots$

Si l'on interprète A comme `line 10`, P comme `turn 25` degrés, et M commme `turn -25 degrés`, l'interprétation de chaînes construit, par raffinements successifs, la « plante » de la figure 4.

2.4 Remarque sur le parcours des itérations

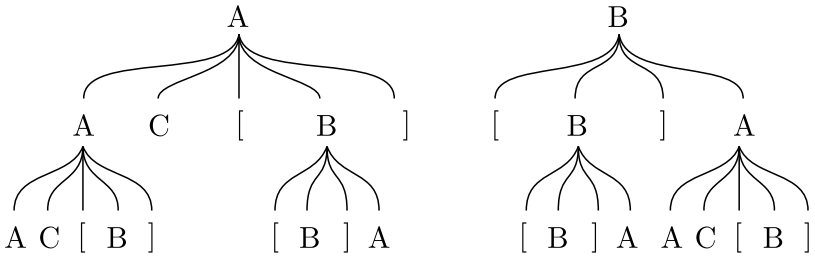
La longueur des itérations successives d'un L-système croît souvent de manière exponentielle. Cependant, le simple parcours d'une itération ne nécessite pas la mise en mémoire de celle-ci. Considérons par exemple le système suivant :

$$G = (\mathcal{S}, w, \sigma) = (\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}, \mathrm{AB}, (\mathrm{A} \mapsto \mathrm{AC[B]}, \mathrm{B} \mapsto \mathrm{[B]A}))$$

L'itération de rang 2 de G est la chaîne :

$$\mathrm{AC[B]C[[B]A][[B]A]AC[B]}$$

La construction de cette chaîne peut être vue comme une forêt dont les feuilles, collectées par parcours gauche, forment exactement la suite des éléments de la chaîne :



L'interprétation des feuilles de ces arbres ne nécessite pas leur mise en mémoire :

- Les « [» ou «] », les symboles qui ne sont pas dans le domaine de σ ainsi que ceux de profondeur 2 font tous partie de la chaîne et peuvent être immédiatement interpétés – puis oubliés.
- Si $X \in \{\mathrm{A}, \mathrm{B}\}$ est rencontré à une profondeur $p < 2$, il suffit d'explorer la forêt associée à la construction de l'itération de rang $p - 2$ de $\sigma(X)$.

3 Implémentation des L-Systèmes en OCaml

3.1 Sujet minimal

Votre travail consistera à écrire puis nous présenter un programme en OCaml permettant au minimum de visualiser les figures produites par des interprétations arbitraires d'itérations de L-systèmes abitraires. Un certain nombre de types, décrits à la section suivante, vous sont imposés. Les autres détails de l'implémentation sont libres, mais il est clair que certains choix seront plus convaincants :

1. Nous vous fournissons un embryon de programme réparti en plusieurs fichiers `.ml` avec leurs fichiers interfaces `.mli`, prêt à compiler en tapant `make`, voir les détails dans le fichier `README.md` du code fourni. Vous pouvez étendre ce programme, ses fichiers, et y ajouter d'autres. Ou bien vous pouvez nous présenter un programme en un seul fichier qu'il faudra invoquer depuis le *toplevel* OCaml (ce qui fait un peu amateur).
2. L'ensemble des L-systèmes et interprétations pourra être câblé en dur dans le programme en suivant l'exemple fourni dans le fichier `Example.ml` (ce qui est très peu flexible) ou mieux, vous pourrez écrire un analyseur permettant à l'utilisateur de rentrer un L-système et une interprétation arbitraires – ou mieux encore, de charger ces données à partir d'un fichier, dont le format pourra s'inspirer des fichiers fournis dans le sous-répertoire `examples/`.
3. Vous pourrez interpréter les chaînes parenthésées en suites de commandes pour la Tortue (ce qui utilisera beaucoup de mémoire) ou mieux, interpréter les chaînes parenthésées à la volée et sans les mettre en mémoire (*c.f.* Section 2.4).
4. Vous pouvez vous servir du module `Graphics` d'OCaml, mais pourquoi pas, d'une autre librairie. Si vous choisissez `Graphics`, nous vous déconseillons de passer trop de temps à simuler une interface graphique avec boutons, menus et boîtes de dialogue — ce n'est pas le sujet du projet.
5. Vous pourrez vous contenter d'afficher le résultat final d'une itération (ce qui est un peu ennuyeux) ou mieux, afficher l'animation correspondant à cette suite.
6. Vous pourrez demander à l'utilisateur l'échelle à laquelle se fait le dessin (ce qui est peu pratique) ou mieux, calculer automatiquement les bornes du graphique pour choisir une échelle raisonnable.
7. Nous vous demandons d'indenter systématiquement votre code, d'éviter les lignes trop longues (pas plus 80 caractères), de choisir des noms parlants pour les variables et fonctions, et de commenter le code souvent, mais en restant concis. Veillez également à supprimer toute duplication de code, suivant le principe DRY ("Don't Repeat yourself").

3.2 Choix des types et code fourni

Les types suivants sont imposés, est il est *obligatoire* de les utiliser dans votre programme. Dans le code fourni, ils sont dans les fichiers `turtle.ml` et `systems.ml`.

Tortue. Le type `command` représente une commande de la Tortue (il faudra bien programmer une tortue) :

```
type command =
| Move of int
| Line of int
| Turn of int
| Store
| Restore
```

Chaînes parenthésées. Le type `'a word` représente l'arbre syntaxique d'une chaîne parenthésée. Ce type est paramétré par une variable `'s` correspondant au type qui sera choisi plus tard pour les symboles – ce pourra être un type algébrique, le type `char`, le type `int`, le type `string` ... en fait, peu importe : les fonctions manipulant des valeurs de type `'s word` ne doivent pas dépendre de ce choix.

```
type 's word =
| Symb of 's
| Seq of 's word list
| Branch of 's word
```

Le constructeur `Seq` représente une suite ordinaire de sous-chaînes. Le constructeur `Branch` représente une chaîne entre crochets. Par exemple, si `a` et `p` sont des variables de type `symbol` (quelle que soit sa définition) la chaîne A[P[PA]A] peut être représentée par

```
Seq [Symb a;
     Branch (Seq [Symb p;
                  Branch (Seq [Symb p; Symb a]);
                  Symb a])]
```

et cette structure aura le type `symbol word`.

Substitutions. Une substitution est représentée par une fonction pure (sans traits impératifs) associant des symboles à des chaînes parenthésées.

```
type 's rewrite_rules = 's -> 's word
```

L'absence d'un symbole dans le domaine d'une substitution sera représentée soit par l'assocation de ce symbole à lui-même, soit par le lancement d'une exception (bien évidemment à capturer).

Interprétations. Une interprétation est une fonction associant des symboles à des listes de commandes pour la Tortue. Ici encore, on se limitera aux fonctions pures.

```
type 's interpretation = 's -> Turtle.command list
```

L-systèmes. Enfin, un L-système est décrit par le type enregistrement suivant :

```
type 's system = {
  axiom : 's word;
  rules : 's rewrite_rules;
  interp : 's interpretation}
```

Exemples de L-systèmes. Nous vous fournissons une petite sélections de L-systèmes (tirés pour la plupart de [1]). Un premier exemple se trouve dans `example.ml`, directement dans le type `system` ci-dessus. D'autres exemples se trouvent dans le sous-répertoire `examples`, dans un format de fichiers fait pour être simple à analyser automatiquement :

- toutes les lignes commençant par # sont des commentaires, ces lignes sont à ignorer.
- le mot sur la première ligne est l'axiome du système
- Après une ligne vide, on trouvera les lignes décrivant les règles du système. Dans une règle, le premier caractère est le symbole concerné, qui sera à réécrire en le mot présent ensuite après un blanc.
- Après une nouvelle ligne vide, on trouvera l'interprétation : de nouveau le symbole concerné est le premier caractère de la ligne, et la suite décrit les commandes correspondante, avec respectivement `M`, `L`, `T`, `S`, `R` correspondant à un `Move`, un `Line`, etc.

Oui il existe des outils en OCaml et autour pouvant aider à l'analyse de fichiers (`Scanf`, `ocamllex`, etc). Mais le format ci-dessus est fait pour pouvoir être lu aisément même sans ces outils. Il faudra juste faire attention au status spécial des caractères `[` et `]`.

3.3 Extensions

Afin de consolider votre note, de nombreuses extensions sont envisageables. Toute extension présentée sera évaluée avec intérêt et indulgence. Si vous êtes à court d'idées, voici quelques suggestions.

Sauvegarde de L-systèmes ou d'images obtenues Nous avons déjà mentionné la possibilité de lire un L-système depuis un fichier texte, soit dans le format proposé ci-dessus, soit dans tout autre format que vous jugerez adéquat. L'opération inverse sera alors une extension possible, à savoir pouvoir écrire un L-système dans un fichier texte au même format. Votre programme pourra être capable d'enregistrer dans un fichier d'image le résultat visuel d'une itération, par exemple sous format *EPS* ou *HP-GL* (facile), ou encore *PDF* ou *SVG* (un peu plus pénible). Si vous décidez d'utiliser une bibliothèque toute faite pour générer le fichier, nous indiquer laquelle. Et assurez-vous d'avoir compris la différence entre une image vectorielle et une image *raster*.

Couleurs. Étendez votre programme pour qu'il génère des figures en couleurs. Pour cela, il faudra étendre le langage de la Tortue avec des commandes permettant de changer la couleur du tracé. Comment faire pour générer des dégradés ?

Variations de paramètres. Que se passe-t'il si on change légerement les interprétations d'angle et/ou de longueurs, en particulier avec les exemples arborescents ? On pourra par exemple contrôler ces variations par des mouvements de souris. Ou utiliser un peu de tirage au hasard.

Graphiques en trois dimensions. Une Tortue qui vole. Une projection orthonormale devrait être facile à implémenter par vous-mêmes, vous n'avez probablement pas envie d'essayer de comprendre OpenGL.

4 Modalités

Le projet est à traiter en groupes de 2 personnes au plus (les projets soumis par des groupes de 3 personnes ou plus ne seront pas acceptés). Les dates limites de soumission et les dates de soutenances seront précisées ultérieurement (a priori début janvier).

Le travail doit être réalisé tout au long de la suite du semestre via un dépôt git pour chaque groupe. L'usage de ce gestionnaire de version (VCS) permet d'éviter toute perte de donnée, et d'accéder si besoin à vos anciennes versions. Ceci permettra nous aussi de garder la trace et les dates de vos contributions, de vérifier la régularité de votre travail et son équilibrage.

– Votre dépôt git doit être hébergé par le serveur GitLab de l'UFR d'informatique :

  `https://gaufre.informatique.univ-paris-diderot.fr`

Voir ci-dessous plus de détail sur son usage. Aucun autre code ne sera pris en compte, même hébergé par un autre serveur.
- Votre dépôt doit être rendu *privé* dès sa création, avec accès uniquement aux membres du groupe et aux enseignants de ce cours. Tout code laissé en accès libre sur Gaufre ou ailleurs sera considéré comme une incitation à la fraude, et sanctionné.
- Il va de soi que votre travail doit être strictement personnel : aucune communication de code ou d'"idées" entre les groupes, aucune "aide" externe ou entre groupes. Nous vous rappelons que la fraude à un projet est aussi une fraude à un examen, passible de sanctions disciplinaires pouvant aller jusqu'a l'exclusion définitive de toute université.

5 Usage de GitLab

La création d'un depôt git sur le GitLab de l'UFR permettra à votre groupe de disposer d'un dépôt commun de fichiers sur ce serveur. Chaque membre du groupe pourra ensuite disposer d'une copie locale de ces fichiers sur sa machine, les faire évoluer, puis sauvegarder les changements jugées intéressants et les synchroniser sur le serveur. Si vous ne vous êtes pas déjà servi de GitLab dans d'autres matières, cette section décrit son usage le plus élémentaire. Pour plus d'informations sur git et GitLab, il existe de multiples tutoriels en ligne. Nous contacter en cas de problèmes.

Accès au serveur et configuration personnelle. Se connecter via l'interface web : `https://gaufre.informatique.univ-paris-diderot.fr`. Utilisez pour cela les mêmes nom et mots de passe que sur les machines de l'UFR, et pas vos compte "ENT" de paris diderot ou u-paris. Cliquer ensuite sur l'icone en haut à droite, puis sur "Settings". A droite, aller ensuite dans la section "SSH Keys", et ajouter ici la partie public de votre clé ssh (ou de vos clés si vous en avez plusieurs). Cela permettra facilitera grandement l'accès ultérieur à votre dépôt git, et vous évitera de taper votre mot de passe à chaque action.

Si vous n'avez pas encore de clé ssh, s'en générer une sur sa machine. L'usage de ssh n'est pas spécifique à git et GitLab, et permet des connections "shell" à des machines distantes. Si vous n'utilisez pas encore ssh et les clés publiques/privées ssh, il est temps de s'y mettre ! Pour plus d'information sur ssh, consulter :

`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/wiki/howto_connect`

Création du dépôt. Pour ce projet, nous vous fournissons quelques fichiers initiaux. Votre dépôt git sera donc un dérivé (ou "fork") du dépôt publique du

cours. En pratique :

1. L'un des membres de votre groupe se rend sur la page du cours
   `https://gaufre.informatique.univ-paris-diderot.fr/letouzey/pf5/`
   s'identifie si ce n'est pas déjà fait, et appuie sur le bouton "fork" (vers le haut, entre "Star" et "Clone"). Attention, un seul "fork" par groupe suffit.
2. Ensuite, aller dans la section "Settings" en bas à gauche, défiler un peu et cliquer sur "Visibility", et selectionner "Private" comme "projet visibility", puis "Save changes" un peu plus bas. Vérifier qu'un cadenas apparaît maintenant à côté de `pf5` quand vous cliquez sur "Projet" en haut à gauche.
3. Toujours dans "Settings" en bas à gauche, mais sous-section "Members" maintenant. "Invitez" votre collègue de projet, ainsi que les logins `aguatto`, `bauer`, `bernardi`, `letouzey`, `padovani` en choissisant "Maintainer" comme rôle.
4. Voilà, votre dépôt sur le GitLab est prêt !

Création et synchronisation de vos copies locales de travail. Chaque membre du projet "clone" le dépôt du projet sur sa propre machine, c'est-à-dire en télécharge une copie locale : `git clone` suivi de l'addresse du projet tel qu'il apparaît dans l'onglet "Clone" sur la page du projet, champ en "SSH". Pour cela, il faut avoir installé `git` et `ssh` et configuré au moins une clé ssh dans GitLab.

Une fois le dépôt créé et cloné et en se plaçant dans le répertoire du dépôt, chaque membre pourra à tout moment :

- télécharger en local la version la plus récente du dépôt distant sur Gitlab :
  `git pull`.
- téleverser sa copie locale modifiée sur GitLab :
  `git push`.

Avant toute synchronisation, il est demandé d'avoir une copie locale "propre" (où toutes les modifications sont enregistrées dans des "commits").

Modifications du dépôt : les commits. Un dépôt Git est un répertoire dont on peut sauvegarder l'historique des modifications. Chaque action de sauvegarde est appelée une *révision* ou "commit". L'*index* du dépôt est l'ensemble des modifications qui seront sauvegardées à la prochaine révision. La commande

`git add` suivi du nom d'un ou plusieurs fichiers

permet d'ajouter à l'index toutes les modifications faites sur ces fichiers. Si l'un d'eux vient d'être crée, on ajoute dans ce cas à l'index l'opération d'ajout de ce fichier au dépôt. La même commande suivie d'un nom de répertoire ajoute à l'index

l'opération d'ajout du répertoire et de son contenu au dépôt. La révision effective du dépôt se fait par la commande

`git commit -m` suivi d'un message entre guillemets doubles.

Invocable à tout instant, la commande

```
git status
```

permet d'afficher l'état courant du dépôt depuis sa dernière révision : quels fichiers ont été modifiés, renommés, effacés, crées, etc., et lesquelles de ces modifications sont dans l'index. Elle indique également comment rétablir l'état d'un fichier à celui de la dernière révision, ce qui est utile en cas de fausse manœuvre.

Les commandes `git mv` et `git rm` se comportent comme `mv` et `rm`, mais ajoutent immédiatement les modifications associées du répertoire à l'index.

Il est conseillé d'installer et d'utiliser les interfaces graphiques `gitk` (visualisation de l'arbre des commits) et `git gui` (aide à la création de commits).

Une dernière chose : git est là pour vous aider à organiser et archiver vos divers fichiers sources. Par contre il vaut mieux ne *pas* y enregistrer les fichiers issues de compilations (binaires, répertoire temporaire tels que `_build` pour `dune`, fichiers objets OCaml `*.cm{o,x,a}`, etc).

Les fusions (merge) et les conflits. Si vous êtes plusieurs à modifier vos dépôts locaux chacun de votre côté, celui qui se synchronisera en second avec votre dépôt GitLab commun aura une manoeuvre nommé "merge" à effectuer. Tant que vos modifications respectives concernent des fichiers ou des zones de code différentes, ce "merge" est aisé, il suffit d'accepter ce que git propose, en personnalisant éventuellement le message de merge. Si par contre les modifications se chevauchent et sont incompatibles, il y a alors un conflit, et git vous demande d'aller décider quelle version est à garder. Divers outils peuvent aider lors de cette opération, mais au plus basique il s'agit d'aller éditer les zones entre `<<<<<` et `>>>>>` puis faire `git add` et `git commit` de nouveau.

Intégrer les modifications venant du dépôt du cours. Si le dépôt du cours reçoit ultérieurement des correctifs ou des évolution des fichiers fournis pour le projet, ces modifications peuvent être intégrés à vos dépôts.

- La première fois, aller dans votre répertoire de travail sur votre machine, et taper :

  ```
  git remote add prof \
  git@gaufre.informatique.univ-paris-diderot.fr:letouzey/pf5.git
  ```

- Ensuite, à chaque fois que vous souhaiterez récupérer des commits du dépôt du cours :

  ```
  git pull prof master
  ```

- Selon les modifications récupérées et les votres entre-temps, cela peut occasionner une opération de "merge" comme décrite auparavant.
- Enfin, ces modifications sont maintenant intégrés à votre copie locale de travail, il ne reste plus qu'à les transmettre également à votre dépôt sur GitLab :

  `git push`

Les branches. Il est parfois pratique de pouvoir essayer différentes choses, même incompatibles. Pour cela, Git permet de travailler sur plusieurs exemplaires d'un même dépôt, des *branches*. Un dépôt contient toujours une branche principale, la branche "master", dont le rôle est en principe de contenir sa dernière version stable. Les autres branches peuvent servir à développer des variantes de la branche master, par exemple pour tenter de corriger un bug sans altérer cette version de référence. La création d'une nouvelle branche, copie conforme de la branche courante – initialement, master – dans son état courant, se fait par :

`git branch` suivi du nom choisi pour la branche.

Sans arguments, cette commande indique la liste des branches existantes, ainsi que celle dans laquelle se trouve l'utilisateur. Le passage à une branche se fait par

`git checkout` suivi du nom de la branche.

Pour ajouter au dépôt distant une branche qui n'est pas encore sur celui-ci, après s'être placé dans la branche :

`git push -set-upstream origin` suivi du nom de la branche

Un `push` depuis une branche déjà sur le serveur se fait de la manière habituelle. Enfin, on peut "réunifier" deux branches avec `git merge`, voir la documentation pour plus de détails.

Noter que GitLab propose également un mécanisme de "Merge Request" : il permet de proposer des modifications, soit à son propre projet, soit au projet qui a été "forké" à l'origine, les membres du projet en question pouvant alors accepter ou non ces suggestions après discussion.

Références

[1] Przemysław Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag. 2004. `http://algorithmicbotany.org/papers/#abop`