université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Projet IPF
LDD2 IM

Point d'étape

Objectif

Le but de cette feuille est de :

1. proposer une façon d'avancer sur le projet sans rester bloqué
2. au delà de ça, montrer une méthodologie pour gérer un **projet** et travailler efficacement en binôme

Cette feuille n'est qu'un guide, en particulier si vous avez réussi à bien avancer sans, vous n'êtes aucunement obligé de vous y conformer. Elle contient cependant des informations utiles pour vous assurer que vous n'avez rien oublié et donne aussi des pointeurs sur ce qu'il faut mettre dans le rapport accompagnant le code.

Remarque : il s'agit d'un **projet** et non pas de TP notés. Des indications trop précises (comme la liste exacte des fonctions à écrire avec leur type) irait à l'encontre de l'esprit de l'UE qui doit laisser une bonne part à votre originalité dans la façon d'aborder le problème. Évidemment, le problème étant relativement guidé, il est normal que tous les groupes qui arrivent à la fin obtiennent une solution similaire, mais certains choix d'implémentation rendront ces projets différents.

Les autres feuilles guides seront moins longues !

1 Plan de bataille

Comme pour tout projet, même si celui-ci est de taille modeste, il convient de lister toutes les tâches à faire. Une tâche, au sens large, est un problème dont la résolution est nécessaire pour l'aboutissement du projet. On dira de plus qu'une tâche est terminale si elle ne peut pas se décomposer en tâches plus simples. Étant donnée une tâche, deux points sont à considérer :

— sa relation par rapport aux autres tâches (de quelles tâches elle dépend et quelles tâches dépendent d'elles) ;
— sa taille : est-il judicieux de la subdiviser en tâches plus petites de façon à pouvoir réfléchir sur chaque sous problème (normalement plus simple que la tâche complète).

Il est aussi important d'avoir une vision *globale* du projet demandé (ici l'écriture d'un programme de résolution de labyrinthes). Pour cela, il faut prendre *du recul* par rapport au sujet. En effet, un sujet doit être rédigé dans un certain ordre qui demande de présenter certains concepts et d'introduire des notations avant de les utiliser. Cependant le sujet ne donne pas nécessaire *le bon ordre* dans lequel aborder les tâches ni la liste de celles-ci. La figure 1 propose un découpage possible du projet en tâches. La figure est organisée comme un arbre et se lit de la gauche vers la droite et du haut vers le bas. Une flèche $\rightarrow$ indique un raffinement d'une tâche en tâches plus précises. Un nœud en **gras** indique une tâche terminale.

Par exemple dans cette figure, la tâche « programme principal » (i.e. l'ensemble du projet) se subdivise en quatre sous-tâches :

— gérer la ligne de commande (qui est en soit une tâche terminale)
— lire et réafficher un labyrinthe
— lire et résoudre un labyrinthe
— générer un labyrinthe aléatoire et l'afficher

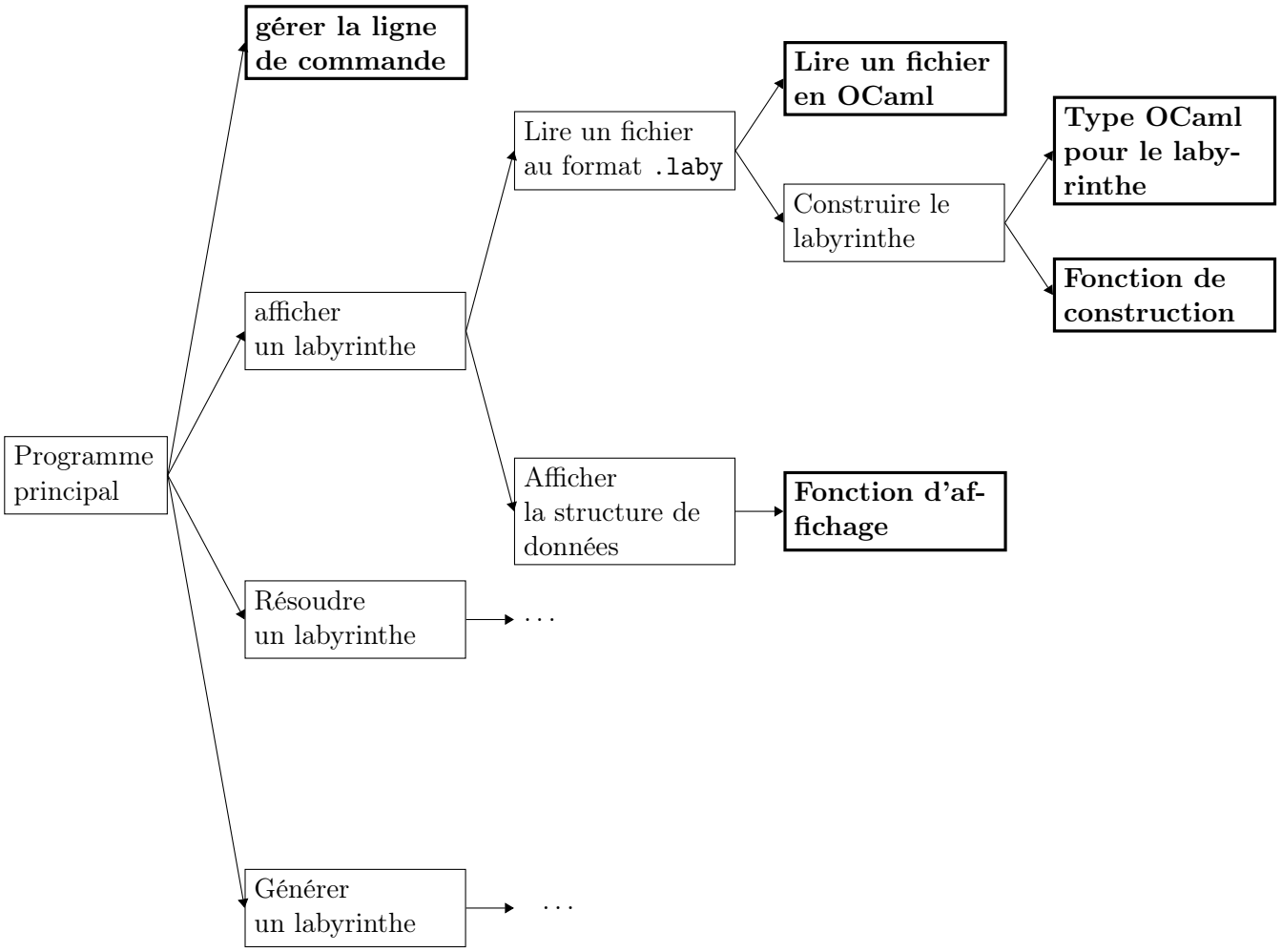

FIGURE 1 – Une décomposition possible du projet en tâches.

La gestion de la ligne de commande constitue une tâche terminale. En effet, il ne semble pas qu'il y ait de concepts à explorer indépendamment au sein de cette tâche. Cette tâche étant terminale, on peut se poser la question de comment la réaliser. Plusieurs choix sont possibles :

— manipulation directe du tableau `Sys.argv` (vivement recommandé) ;
— utilisation du module `Arg` d'OCaml (cf. la documentation de la bibliothèque standard, peut être fait dans un deuxième temps pour découvrir ce module).

Il est aussi important de se poser la question de la *complétion de la tâche*. Dans le cadre d'un programme informatique, c'est relativement simple : une fois une tâche terminée, le programme que l'on essaye d'écrire doit faire plus de choses que ce qu'il faisait avant (ou faire le même nombre de choses mais les faire plus efficacement, ...).

Pour l'exemple de la lecture de la ligne de commande, avant la réalisation de la tâche, on n'a un programme qui ne fait rien une fois lancé (ou exécute du code de test). Après la réalisation de la tâche, le programme doit pouvoir gérer les scénarios suivants :

1. `./maze.exe --help` : affiche un message d'aide sur les différentes options
2. `./maze.exe print fichier.laby` : affiche un message `Affichage du labyrinthe à faire!`
3. `./maze.exe solve fichier.laby` : affiche un message `Résolution à faire!`
4. `./maze.exe random 10 10 42` : affiche un message `Génération aléatoire à faire!`

De plus, les erreurs qui sont gérables à ce niveau doivent l'être :

— option non reconnue
— nombre de paramètres sur la ligne de commande invalide
— existence des fichiers passés sur la ligne de commande

En cas d'erreur, on peut terminer le programme par un code de sortie non nul, différent pour chaque erreur. Les modules et fonctions qui peuvent être utiles sont :

— `exit : int -> 'a` : fonction qui termine le programme avec le code de sortie donné en argument

— Celles du module et `Filename` : module de la bibliothèque standard permettant de manipuler les noms de fichier, extension, chemin, ...

Remarque : tous les choix faits ici sont laissés libres, mais méritent d'être présentés succinctement dans le rapport final.

2 Indications

2.1 Ligne de commande

La lecture de la ligne de commande ne pose pas de problème conceptuels (il s'agit juste, étant donné un tableau de chaînes de caractères de voir s'il y est de la bonne forme). Une fonction qui peut être intéressante est la fonction `Array.to_list` permettant de convertir un tableau en liste. Une fois cette fonction utilisée, et comme on a la garantie que le tableau `Sys.argv` contient toujours dans la case `0` le nom de l'exécutable, on peut écrire du code comme :

```
1   match Array.to_list Sys.argv with
2     [ _; "--help" ] -> display_help_message ()
3   | [ _; "print"; fichier ] -> read_and_print fichier
4   | ... (* tous les autres cas *)
5   | _  -> (* ligne de commande invalide *)
```

2.2 Lecture de fichier

On a donné dans la feuille 1 une façon de lire un fichier ligne à ligne. On développe un peu avec quelques remarques. Pour lire une ligne de texte d'un fichier ouvert, on utilise `input_line c_in` (où `c_in` a été obtenu avec `open_in "fichier.laby"` par exemple). Si le but est de réafficher immédiatement chaque ligne, on n'a pas de question à se poser. Si le but est plutôt de collecter toutes les lignes par exemple dans une liste, alors la situation est (un peu) plus complexe. En effet, on doit faire une fonction récursive qui « boucle » pour chaque ligne du fichier jusqu'à obtenir l'exception `End_of_file`. La fonction ci-dessous pourrait faire l'affaire :

```
1 let read_file c_in =
2   let rec loop acc =
3     try
4       let s = input_line c_in in
5       loop (s :: acc)
6     with
7       End_of_file -> List.rev acc (* accumulé à l'envers *)
8   in
9   loop []
```

Le problème est que la fonction interne n'est pas récursive terminale. Ce n'est pas gênant pour notre projet, car les fichiers de labyrinthes n'auront au plus que quelques centaines de lignes, mais ce code ne devrait pas être réutilisé pour lire des fichiers arbitraires avec des dizaines de milliers de lignes, car il fera un « Stack overflow ». Une façon de contourner le problème est d'utiliser un *motif d'exception* dans un filtrage. En effet, la construction `match ... with` montrée en cours d'IPF est un peu plus complexe et à la forme suivante :

```
  match e with
     m1  -> e1
   | ...
   | mn  -> en
   | exception Err1 -> err1
   | ...
   | exception Errm -> errm
```

Cela signifie, que l'expression `e` est évaluée puis comparée tour à tour aux différents motifs (`m1`, ..., `mn`) et la « branche » correspondant au premier motif satisfait est évaluée. Cependant, s'il l'évaluation de `e` provoque une erreur, alors si celle-ci est l'une de celles listées dans les branches **`exception`** `Err` alors la branche correspondante est évaluée, sinon l'exception est propagée plus haut dans le programme. On peut donc ré-écrire la fonction `read_file` comme :

```
let read_file c_in =
  let rec loop acc =
    match input_line c_in with
      s -> loop (s :: acc)
    | exception End_of_file -> List.rev acc (* accumulé à l'envers *)
  in
  loop []
```

2.3 Labyrinthe

On propose, comme indiqué dans la feuille 1 de séparer dans un module la structure de donnée du labyrinthe. On doit donc ajouter deux fichiers

— `grid.ml` contenant le code permettant de représenter une grille, la lire, l'afficher, la manipuler, ...

— `grid.mli` la signature des fonctions

Le fichier `grid.mli` pourra contenir :

```
type t
(** Le type des grilles de labyrinthe *)

val from_file : in_channel -> t
(** [from_file c_in] lit les lignes se trouvant dans le
    descripteur de fichier [c_in] et construit une grille.
    Peut lever une exception si le fichier n'est pas au bon format.
  *)

val print : t -> unit
(** [print g] affiche la grille [g] sur la sortie standard. *)

....
```

Les commentaires commençant par `(**` sont des commentaires de *documentation*. L'éditeur `VSCode` sait utiliser ces commentaires lorsque l'on survole une fonction et permet d'en afficher l'aide. C'est particulièrement précieux car vous aller être amené à utiliser des fonctions écrites par quelqu'un d'autre (votre binôme).

Le fichier **grid.ml** doit lui contenir des définitions concrètes, notamment la définition du type pour représenter vos grilles. Vous devez donc y réfléchir en vous posant les questions suivantes.

- quelles informations vont être stockée dans la grille ? Vous devez stocker suffisamment d'information pour pouvoir ré-imprimer la grille à l'identique, donc les informations de murs, de case de départ et de case d'arrivée semblent un minimum
- comment l'objet grille va-t'il être manipulé ? Même sans connaître les algorithmes que vous allez utiliser, il semble raisonnable de penser que :
  - on va se déplacer de case en case (donc, comment identifier une case parmi d'autres ?)
  - étant données une case on veut connaître les autres cases accessibles
  - on veut savoir si on est arrivé
  - on va construire des labyrinthes, donc on doit pouvoir créer une grille « vide » ou « pleine » de dimensions données, puis pouvoir rajouter ou supprimer des murs

Ces questions vont impacter la représentation interne des grilles et les fonctions exportées dans le fichier **grille.mli**. Le fichier **grille.ml** va donc contenir du code comme :

```
1  type t = int list list (* Exemple ! ne pas réutiliser tel quel ! *)
2
3  let read_file c_in = ... (* vu précédemment * )
4
5  let from_file c_in =
6    let lines = read_file c_in in (* on a la liste des lignes *)
7    (* on fait quelque chose avec lines pour créer la grille *)
8
9
10 let print g =
11   List.iter (fun l -> ... ) g
12   (* là encore un exemple *)
```

2.4 Comment travailler ?

Découpage en tâches : le découpage en tâche est un travail complexe. Il devient de plus en plus simple avec l'expérience (mais comme on est confronté à des projets de plus en plus complexes cela reste une chose difficile). Ce n'est pas non plus un processus figé : il est tout à fait courant de se rendre compte, en programmant, qu'une tâche que l'on pensait terminale a besoin de sous-tâches. On peut donc prendre un moment pour mettre à jour son schéma des tâches (sur une feuille ou dans sa tête). On peut en particulier se demander si ces nouvelles tâches sont réutilisables par d'autres (ce qui entraînera une factorisation de code).

Répartition du travail : le but d'un projet en binôme est de favoriser le travail collaboratif. Il faut cependant éviter trois mauvaises situations :

- un membre du binôme fait tout et l'autre ne fait rien ;

— les deux membres du binôme font tout en même temps. C'est une perte de temps : certaines tâches doivent se faire en parallèle sous peine de manquer de temps pour finir le projet.
— les deux membres du binôme découpent sommairement le projet en deux et font chacun la moitié de bout en bout : très risqué, réunir deux morceaux de code conçus en totale isolation est assez compliqué.

On doit donc alterner des séquences de travail à deux (choix des tâches, réflexions algorithmiques, choix des structures de données) avec des séquences de travail individuel (ex : l'une écrit le code de `grid.ml` pendant que l'autre travaille sur la ligne de commande). La collaboration est d'autant plus simplifiée que l'on s'est mis d'accord en amont sur des types, des noms de fonctions etc.

Écriture du code et tests : pour un projet de cette envergure, il est vivement recommandé de compiler **souvent** (`dune build`) et de penser très tôt à des cas de tests. L'utilisation de Visual Studio Code peut aider car les erreurs de typage sont indiquées en temps réel. Le programme étant basé essentiellement sur *des fichiers*, il est primordial de conserver **TOUT** les fichiers de tests utilisés dans un répertoire `test/`. Ces derniers devront être mentionnés dans le rapport.