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.

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 compression/décompression de textes). 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 nécessairement *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

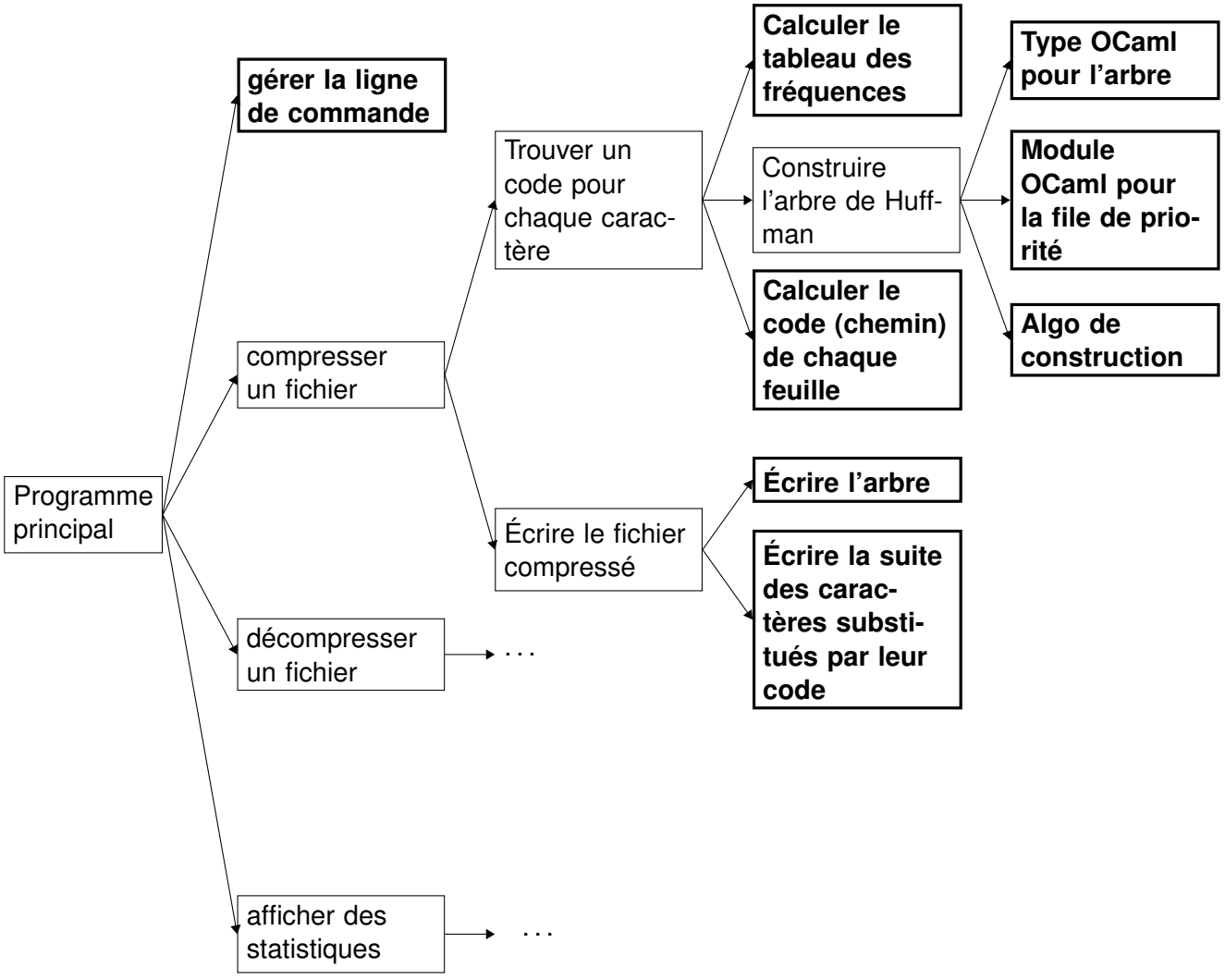

FIGURE 1 – Une décomposition possible du projet en tâches.

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 quatre sous-tâches :

— gérer la ligne de commande (qui est en soit une tâche terminale)

— compresser un fichier

— décompresser un fichier

— afficher des statistiques

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 indépendemment 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`;

— utilisation du module `Arg` d'OCaml (cf. la documentation de la bibliothèque standard).

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 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. `huff --help` : affiche un message d'aide sur les différentes options
2. `huff fichier` : affiche un message `Compression du fichier à faire!`
3. `huff fichier.hf` : affiche un message `Décompression du fichier à faire!`
4. `huff --stats fichier` : affiche un message `Compression du fichier et stats 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 `Arg` (optionnel, gestion de la ligne de commande), `exit : int -> 'a` (fonction qui termine le programme avec le code de sortie donné en argument), `Sys` et `Filename` (modules de la bibliothèque standard permettant de tester les fichiers et de manipuler leur nom, 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 Arbre de Huffman

Un point central de ce projet est l'arbre de Huffman, présenté dans le sujet. Comme l'algorithme de compression repose sur la construction d'un tel arbre, il convient de se poser la question de son implémentation en OCaml. Conceptuellement, l'arbre de Huffman est très simple :

— c'est un arbre binaire, c'est à dire que chaque nœud possède soit deux fils (nœud interne) soit aucun (feuille)

— un nœud interne ne contient que ses deux sous-arbre et pas de donnée associée

— une feuille contient un caractère

En OCaml, une définition naturelle pour de tels arbres est :

```
1   type tree =
2       | Leaf of int
3       | Node of tree * tree
```

L'entier stocké dans la feuille correspond au code du caractère. On peut alternativement stocker des valeurs du type `char`. Cependant, les caractères étant parfois utilisés en tant qu'indices de tableau (par exemple le tableau des fréquences de chaque caractère), stocker un entier plutôt qu'un `char` évite d'alourdir le code avec des conversions entre ces deux types.

2.2 File de priorité

L'algorithme de création de l'arbre repose de façon implicite sur une structure de données : la file de priorité. En effet, la file est initialement remplie avec des paires : `(n, Leaf c)` où `n` est le nombre d'occurrences du caractère de code `c` dans le fichier.

L'algorithme répète ensuite l'opération consistant à extraire de la file les deux paires dont la première composante sont les plus petites (par exemple `(n1, t1)` et `(n2, t2)`), et de recréer une nouvelle paire `(n1 + n2, Node(t1, t2))`, et remettre celle ci-dans la file.

Il faut donc compléter le fichier `heap.ml` de façon à obtenir une structure de file de priorité. Les fonctions exportées par ce module (celles listées dans `heap.mli`) peuvent ensuite être utilisées.

2.3 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 entrainera 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 `heap.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** 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. Pour les tests, il est recommandé d'écrire des fonctions auxiliaires. Par exemple, une fonction qui affiche

une forme lisible de l'arbre de Huffman dans la console peut permettre d'augmenter sa confiance dans le code écrit. Par exemple, vous pouvez afficher l'arbre que vous calculez pour le mot "satisfaisant" et vérifier qu'il est similaire à celui de l'énoncé.