Projet PAF 2020: *Dungeon Crawling*

PAF 2020 - Master STL S2

Version du 17/05/2020

Ce document présente le sujet du projet de l'UE PAF 2020 et contient les attendus, la description du projet, un guide pour démarrer (qui peut ne pas être suivi), une liste des extensions possibles et des consignes pour le rapport et la soutenance.

Modification de la version du 17/05:

- deadlines (1.2 et 1.3)
- consignes de rendu final (4.1)
- property-based testing (4.2)
- classes algébriques (4.3)
- suppression des mentions à la soutenance (6)
- barême prévisionnel modifié (6)

1 Objectifs et Rendus

1.1 Objectifs

L'objectif principal de ce projet est le développement d'un **jeu** vidéo **sûr**, dans le langage **Haskell**. L'évaluation portera sur ces deux points:

1. programmation **sûre**: les types et les fonctions associées du projet doivent toutes être accompagnées de propositions (fonctions `-> Bool`) d'**invariants**, de **pré-conditions**, et de **post-conditions**. Les

opérations des entités doivent toutes être testées en utilisant, de façon raisonnablement complète, les **tests unitaires** (*HSpec*) et les **tests basés sur les propriétés** (*Quickcheck*).

2. programmation **fonctionnelle**: le langage Haskell est imposé, l'application doit être écrite dans un style fonctionnel, une attention doit être donnée à la séparation de la logique (évolution de l'état du jeu) et les effets de bords (boucle de jeu, entrée/sortie, affichage). L'utilisation de spécificités du langage Haskell est encouragée, mais pas nécessaire: types définis comme sommes d'états remarquables, instanciation de classes de types, utilisation des structures algébriques (`Functor`, `Applicative`, `Monad`).

Le projet sera évalué sur ces deux critères, ce qui veut dire, entre autres;

- que si elles sont appréciées par les correcteurs, les extensions/fonctionnalités superficielles (rendu graphique évolué, gestion du son, interface utilisateur) qui ne contribuent pas aux deux objectifs, ne seront pas prises en compte dans la notation.
- que le projet peut s'éloigner des caractéristiques proposées dans ce document, tant qu'il satisfait les deux critères.

1.2 Premier Rendu

Le projet est à faire en binôme (monôme possible avec autorisation de l'équipe enseignante, trinôme non-autorisé). On utilisera le *resolver* `lts-12.26` comme en TME, si possible.

Le premier rendu consiste en un projet GitLab. Les enseignants doivent y être ajoutés comme *maintainer*. La deadline est le **13 Mai à 23h59**.

Le premier rendu doit contenir le jeu de base.

1.3 Rendu Final

La deadline de **rendu final** du projet est **le Mercredi 10 juin à 23h59**. Les enseignants doivent être passé en *owner*.

Le rendu final doit comporter le rapport.

2 Description du Jeu

L'objectif de ce projet est la réalisation d'un *dungeon crawler* (`https://fr.wikipedia.org/wiki/Dungeon_crawler`): un jeu vidéo simple basé sur le paradigme *porte/monstre/trésor*. Le joueur déplace son personnage dans un labyrinthe figuré par une grille discrète (i.e. les déplacements se font case par case): le labyrinthe contient des monstres, des pièges, des portes et des trésors. L'objectif du joueur est de trouver la sortie, ou de rapporter à la sortie un trésor particulier. Les monstres sont autonomes, leur comportement (déplacement, interaction avec le joueur) est contextuel (il peut dépendre de ce qui se trouve autour du monstre).

2.1 Fonctionnalités

- le jeu doit permettre le chargement de labyrinthes importés sous forme de fichiers texte (mais on n'impose pas un standard pour ces fichiers),
- l'état du jeu (labyrinthe, joueur, monstres) doit être affiché à l'aide d'une interface graphique (en SDL2), qui représente (entre autres) le labyrinthe en "vue du dessus",
- le joueur doit pouvoir diriger son personnage dans les quatres directions (Nord, Est, Sud, Ouest) au clavier,
- le joueur doit pouvoir interagir avec le labyrinthe et son contenu (ouvrir des portes, combattre des monstres, ramasser des objets).
- le jeu doit avoir des conditions de victoire (ramasser un trésor, trouver la sortie du labyrinthe, vaincre tous les montres, ...) et de défaite (perdre toute sa vie, tomber dans un piège mortel, ...) atteignables.

- la *boucle de gameplay* du jeu peut être construite à partir de celle du TME6.

2.2 Inspiration

Voici une sélection de jeux vidéo dont on pourra s'inspirer (l'objectif n'est évidemment pas de faire un *remake*, et les fonctionnalités originales sont encouragées):

- Version **classique**:
  - *Hack* (`https://en.wikipedia.org/wiki/Hack_(video_game)`)
  - *Crawl* (`https://en.wikipedia.org/wiki/Dungeon_Crawl_Stone_Soup`)
  - *ToME* (`https://en.wikipedia.org/wiki/Tales_of_Maj%27Eyal`)
- Version **première personne**:
  - *Dungeon Master* (`https://en.wikipedia.org/wiki/Dungeon_Master_(video_game)`)
  - *Legend of Grimrock* (`https://en.wikipedia.org/wiki/Legend_of_Grimrock`)
  - *Scarab of Ra* (`https://en.wikipedia.org/wiki/Scarab_of_Ra`)
- Version **continue**:
  - *Gauntlet* (`https://en.wikipedia.org/wiki/Gauntlet_(1985_video_game)`)
  - *Enter the Gungeon* (`https://en.wikipedia.org/wiki/Enter_the_Gungeon`)

3 Guide

Dans cette section, on propose des guides pour programmer une première version du moteur de jeu. Ces guides donnent des pistes d'implémentation pour les composants de base du jeu, mais peuvent être complétement ignorés par les binômes à l'aise en programmation fonctionnelle (c'est-à-dire que le respect de ces guides n'est pas pris en compte dans l'évaluation du projet). Autrement dit, toutes les choses présentées dans les guides sont des **suggestions**.

3.1 Guide: Terrain de Jeu

Le terrain de jeu ("niveau") est une carte rectangulaire de cases. Dans une première version, les cases d'une carte pourront être des cases vides (franchissables par le joueur et les monstres), des murs (non-franchissables), des portes (elles ont une orientation Est-Ouest ou Nord-Sud et peuvent être ouvertes ou fermées), des entrées (l'endroit par lequel le joueur entre dans le labyrinthe) et des sorties (l'endroit par lequel le joueur sort du labyrinthe).

Une manière de décrire une carte est une table d'association entre des coordonnées (abscisses, ordonnées) et des cases (cf. Figure 1) qui seront définies dans un module `Carte.hs`.

A partir de ce type, on peut déjà faire trois choses intéressantes;

1. instancier `Show` and `Read` avec `Carte` pour pouvoir manipuler (lire et écrire) des fichiers texte représentant les cartes (cf. exemple de la Figure 2, les `X` sont des murs, les `-` et `|` des portes fermées, ...). Comme les cartes contiennent un `Map Coord Case`, il faut instantier `Ord` avec `Coord` d'une manière cohérente avec les opérations de lecture et d'écriture (on lit un "fichier carte" ligne-par-ligne).
2. proposer un **invariant** pour les cartes qui permet d'identifier des cartes *saines*. On peut exiger par exemple, pour une carte saine:
   - que les clefs de sa table d'association soient des coordonnées qui se trouvent dans le rectangle décrit par sa hauteur et sa largeur,
   - que chaque coordonnées qui se trouve dans le rectangle décrit par sa hauteur et sa largeur ait une valeur associée dans la table,
   - qu'elle comporte une unique entrée et une unique sortie,
   - qu'elle soit entièrement entourée de murs,
   - que chaque porte soit encadrée par deux murs,
   - que la sortie soit accessible depuis l'entrée (à une entité qui sait ouvrir les portes).

```haskell
import qualified Data.Map.Strict as M

data PDirection = NS | EO deriving Eq -- direction d'une porte

data StatutP = Ouverte | Fermee deriving Eq -- statut d'une porte

data Case = Normal -- une case vide
    | Porte PDirection StatutP -- une porte ouverte ou fermee
    | Mur -- infranchissable (sauf pour les fantomes ...)
    | Entree -- debut du niveau
    | Sortie -- fin du niveau
    deriving Eq

data Coord = C {cx :: Int, cy :: Int} deriving Eq

data Carte = Carte {cartel :: Int, -- largeur
                    carteh :: Int, -- hauteur
                    carte_contenu :: (M.Map Coord Case) -- cases de la carte
                    }
```

Figure 1: Type des Cartes (Suggestion).

```
XXXXXXXXXX
X    | XSX
X    X X-X
XXXX X X X
X    X X X
X XXXX   X
X    XXXXX
XXX    XXX
XE  X  XXX
XXXXXXXXXX
```

Figure 2: Exemple de contenu d'un fichier texte représentant un labyrinthe 10x10

Il convient de transformer chaque "règle" de l'invariant en une propriété, puis d'écrire des tests vérifiant ces propriétés.

3. énumérer des **opérations** sur les cartes permettant, par exemple:
   - d'**observer** la case correspondant à des coordonnées,
   - d'**observer** si une case est franchissable (pour quelqu'un qui sait, ou non, ouvrir les portes),
   - d'**opérer** une édition d'une case,
   - d'**opérer** l'ouverture/la fermeture d'une porte.

   Il faut pour chaque opération, ajouter une **pré-condition** si elle est nécessaire et une **post-condition**, et écrire des tests pour ces propriétés.

3.2 Guide: Entités

On appelle *entité* les choses qui peuvent se trouver dans dans un labyrinthe, cela inclue le(s) joueur(s), les entités mobiles (appelée *mob*) comme les montres ou les projectiles, et les objets (trésors, clefs, potions, pièces d'équipements, ...) que le joueur peut ramasser.

Le type des entités doit se définir comme une somme de produits, décrivant les différentes entités que le jeu peut manipuler. Dans ce guide, une entité est soit le joueur, soit une vache (un type de monstre).

Les entités doivent disposer d'un "champ" `iden` qui permettra à la logique du jeu d'identifier uniquement une entité. Le joueur et les vaches possèdent un nombre de "points de vie" (si on décide d'implémenter le combat).

```
data Entite = Vache {iden :: Int, pvie :: Int}
            | Joueur {iden :: Int, pvie :: Int}
            deriving (Eq)
```

3.2.1 Environnement

Le rôle d'un *environnement* est de décrire la position des entités actuellement présentes dans le labyrinthe à l'aide d'une table d'associations (les clefs sont des coordonnées et les valeurs des listes d'entités).

```
data Envi = Envi {contenu_envi :: M.Map Coord [Entite]}
```

Ainsi, si `lookup (C x y) (contenu_envi env)` s'évalue en `Nothing` (ou `Just []`, mais on pourra éviter ce cas avec un invariant), cela veut dire qu'aucune entité ne se trouve en dans la case (x, y), et si cela s'évalue en `Just l`, alors `l` est la liste de toutes les entités contenues en (x, y).

Pour manipuler un environnement on peut implémenter les opérations suivantes (encore une fois, c'est une suggestion):

- `franchissable_env :: Coord -> Envi -> Bool` qui décide si une case contient ou non des entités infranchissable (on veut que les monstres puissent se bloquer entre eux)
- `trouve_id :: Int -> Envi -> Maybe (Coord, Entite)` qui cherche dans un environnement une entité depuis son identifiant et renvoie, si elle existe, un couple correspondant à l'entité et aux coordonnées auxquelles elle se trouve.
- `rm_env_id :: Int -> Envi -> Envi` qui retire d'un environnement une entité depuis son identifiant.
- `bouge_id :: Int -> Coord -> Envi -> Envi` qui déplace une entité au sein d'un environnement, depuis son identifiant.

On écrira, bien évidemment, écrire un invariant pour le type `Environnement` et des pré-conditions et post-conditions pour les opérations, ainsi que des tests associés.

3.2.2 Modèle

Le moteur de jeu va maintenir une structure de données d'entités (joueur, monstres, objets) qui va évoluer de la manière suivante: à chaque **tour** de jeu (chaque appel récursif de la *boucle de gameplay*), on appelle **successivement** la méthode `tour` de chaque entité, qui effectue une ou plusieurs actions (se déplacer, attaquer, ...).

Pour calculer la méthode **tour** d'une entité, on a besoin de connaître le **modèle** du jeu dans lequel se trouve cette entité, c'est-à-dire, la carte et l'environnement. Deplus, la méthode **tour** d'une entité peut produire des effets sur ce modèle du jeu (ouverture d'une porte de la carte, déplacement d'une entité de l'environnement). On définit les modèles ainsi (on insiste sur le fait qu'il s'agit d'une suggestion initiale, qui pourra être modifiée au fur et à mesure de l'avancée dans le projet):

```
data Modele = Cont {carte :: Carte, -- carte actuelle
                    envi :: Envi, -- environnement actuel
                    gene :: StdGen,  -- generateur aleatoire
                    log :: String, -- journal d
                    keyboard :: Keyboard, -- l'etat du clavier
                    }
```

Ainsi un élément du type `Modele` contient une carte, un environnement, un générateur aléatoire (nécessaire pour pouvoir traiter correctement les comportements non-déterministes), une chaîne de caractères (correspondant à un "journal du tour", qu'on pourra afficher sur la sortie standard) et un état du clavier (cf. TME6).

Maintenant on peut définir `bouge :: Modele -> Entite -> Coord -> Modele` qui déplace une entité dans un modèle (en mettant à jour l'environnement Modèle), si elle existe, avec les préconditions, postconditions adéquates.

3.2.3 Ordres

Pour simuler une "intelligence" chez les *mobs* autonomes (qui ne sont pas contrôlés directement par l'utilisateur) on décompose l'opération `tour` en deux opérations:

- *prévoir*, qui consiste à générer une liste pondérée d'**ordres**.
- *décider*, qui prend un ordre au hasard dans la liste pondérée (en fonction des poids) et l'applique.

Voici une première énumération possible pour les ordres (qui sont partagés par toutes les entités).

```
-- N : Aller en haut
-- S : Aller en bas
-- E : Aller a droite
-- O : Aller a gauche
-- U : Utiliser (contextuel)
-- R : Ne rien faire
data Ordre = N | S | E | O | U | R deriving Show
```

On peut écrire une fonction `decide :: [(Int, Ordre)] -> Modele -> Entite -> Modele` qui prend une liste pondérée d'ordres, tire un ordre au hasard, et l'applique a une entité accompagnée d'un modèle. En effet, la plupart des ordres (se déplacer, ne rien faire) seront communs à toutes les entités. D'autres ordres (`U`) dépendent de l'entité à qui ils sont appliqués (par exemple, `U` peut permettre à un joueur d'ouvrir une porte).

3.2.4 Vaches

On donne ici un unique exemple d'entité autonome: les vaches. Les vaches sont des monstres (au sens où, ce sont des objets mobiles différents du joueur). Elles sont pacifiques et se déplacent aléatoirement dans le labyrinthe. Elle ne peuvent pas ouvrir une porte, ni se déplacer dans une case qui contient un autre *mob*. Leur comportement est tres simple le tour d'une vache consiste uniquement à bouger (ou non, elles peuvent décider de ne rien faire) dans une case adjacente libre.

On pourra écrire une fonction `prevoit_vache :: Modele -> Entite -> [(Int, Ordre)]` qui prend une vache et le modèle du jeu et fabrique la liste pondérée des ordres possibles (par exemple, "ne rien faire avec un poids 2, aller au Nord avec un poids 1, aller au Sud avec un poids 1" pour une vache qui serait actuellement incapable de bouger à l'Ouest ou à l'Est - parce qu'elle est bloquée par des murs, par exemple)

Comme toujours, les types et les fonctions doivent s'accompagner de propositions correspondant à des invariants, pre-conditions et post-conditions adéquats.

3.2.5 Joueur

Le joueur est l'autre exemple initial d'entité. Il se déplace de manière déterministe en réagissant à la pression éventuelle d'une touche de direction sur le clavier (en regardant le `Keyboard` du `Modele`, comme dans le TME6). Il ne peut pas se déplacer dans une case infranchissable (entre autres, parce qu'elle contiendrait un autre *mob*). Une touche spéciale (barre d'espace) permet d'ouvrir/fermer une porte (on pourra, par exemple, faire que l'utilisation de cette touche change l'état de toutes les portes adjacentes au joueur, pour éviter d'avoir à "viser" une porte).

3.3 Guide: Moteur de Jeu

Le moteur de jeu doit maintenir un *état du jeu*, qui doit contenir les informations qui persistent à travers les tours:

```
data Etat = Perdu
        | Gagne
        | Tour {num_tour :: Int,
                carte_tour :: Carte,
                envi_tour :: Envi,
                gen_tour :: StdGen,
                obj_tour :: (M.Map Int Entite),
                journal_tour :: String}
```

Un état qui n'est ni perdu ni gagné contient le numéro du tour courant, la carte courante, l'environnement courant, le générateur aléatoire courant, une table d'association contenant les entités de l'environnement (on vérifiera avec un invariant que les deux sont cohérents) et une chaîne correspondant au "journal du tour" (qu'on affichera sur la sortie standard à chaque tour).

Une fonction `etat_tour :: Etat -> Keyboard -> Etat` (on peut ajouter un `a` instanciant `RealFrac` si on veut passer le `deltaTime`, comme dans le TME6, si on en a besoin) s'occupe de prendre un état et de calculer l'état correspondant au tour suivant, elle le fait en appelant **successivement** la fonction `tour` de toutes les entités de `obj_tour`, en transportant un modèle du jeu. Le modèle du jeu récupéré à la fin de cette opération permet de créer le nouvel état du jeu.

On devra aussi gérer les notions de **victoire** et de **défaite**. Initialement, le jeu est gagné si le joueur se trouve sur la case de sortie (et il n'y a pas de conditions de défaite)

Remarque: Dans les *dungeon crawlers* classique, la boucle de gameplay est réactive: le jeu ne progresse pas automatiquement. Sans interaction du joueur, l'état du jeu reste figé (ce qui permet à l'utilisateur de prendre le temps de réfléchir à sa prochaine action) et c'est seulement quand le joueur entre une commande qu'on fait progresser le jeu d'un tour. Pour le projet, on pourra choisir d'utiliser une boucle réactive ou non. Par défaut, on prendra une boucle **non-réactive** (il est possible, mais pas certain, que ce soit plus simple), qui avance automatiquement avec le temps machine (comme en TME6), mais c'est un point laissé entièrement **libre**?

3.4 Main

Le fichier `Main.hs` contiendra la *boucle de gameplay*, qui sera similaire à celle du TME6 et qui appellera à chaque tour la fonction `etat_tour` pour calculer le nouvel état (en plus de s'occuper de la gestion de l'affichage, du temps, et des évènements clavier, comme au TME6).

Il faut aussi inclure dans la fonction `main` le chargement des *sprites* du jeu et un système minimal d'initialisation (qui crée un état initial à partir d'un fichier texte représentant la carte, en plaçant le joueur sur l'entrée, et quelques vaches dans le labyrinthe).

4 Rendu Final et Extensions

4.1 Consignes

Pour obtenir une **note satisfaisante** à l'évaluation, il faut que le projet respecte les contraintes suivantes:

- implémentation du **jeu de base** (Section 3) en Haskell + SDL2 (ou une modification du jeu de contenu équivalent),
- rédaction de **propriétés** pre/post/inv pour les types et les opérations du jeu,
- test systèmatique des propriétés pre/post/inv incluant **au moins une utilisation pertinente de test basé sur les propriétés** (*Quickcheck*).
- quelques extensions basiques, afin de proposer un contenu ludique minimal.

Pour obtenir **la note maximale**, le projet doit inclure de multiples extensions parmi celles proposées ou non (c'est encore mieux si elles sont originales), une utilisation pertinente **du test basés sur les propriétés** et **des classes algébriques**.

La motivation des extensions est importante: il faudra montrer ce qu'elles apportent au jeu, mais surtout ce qu'elles apportent au projet: complexité de la rédaction de propositions, élégance du code fonctionnel

utilisé, défi pour obtenir des tests complets, ... Une extension qui ne contiendrait pas de contenu "logiciel sûr" (propositions et tests) ou de contenu "programmation fonctionnelle" (utilisation de construction propres à la programmation fonctionnelle ou à Haskell pour la mettre en oeuvre) n'ajouterait rien à l'évaluation (par exemple, inclure plusieurs ennemis différents dont les comportements seraient similaires et dont seules les statistiques différeraient).

4.2 Tests Basés sur la propriété

Le projet doit contenir au moins une utilisation pertinente de *QuickCheck*, conforme à celles proposées en TME. Un soin particulier sera apporté à l'efficacité des générateurs (cf. le sujet de TME sur l'utilisation de `suchThat`).

La rédaction et l'utilisation de générateurs complexes pour des types non-triviaux est fortement encouragée. Dans tous les cas, les tests doivent être décrits dans le rapport et la méthode de rédaction des générateurs doit y être explicitée.

4.3 Algébriques

Les projets doivent rendre explicite la maîtrise des classes algébriques (vues dans les Cours/TDs 06-09) par leurs auteurs:

- manipulation correcte, élégante et efficace de `Maybe`, `Either`, `IO`, ...,
- instanciation de classes algébriques par des types définis dans le projet (par exemple `instance Monad EtatDuJeu where ...`) et utilisation des bénéfices apportés par l'instanciation (primitives, *do-notations*, compositions, ...)

La programmation "avancée" avec les algébriques (rédaction de *transformers*, ...) est attendue pour une évaluation maximale.

4.4 Extensions: Intérêt ludique

Voici quelques extensions renforçant l'intéret du projet en tant que "jeu vidéo".

4.4.1 Combats

Le joueur peut attaquer les monstres qui lui sont adjacents (par exemple, en essayant de se déplacer sur leur case, ou en invoquant une commande spécifique). Cette extension reste facultative pour permettre aux binômes qui le souhaitent de faire un jeu non-violent.

4.4.2 Ennemis dangereux

Des ennemis qui peuvent attaquer le joueur: ils sont capable de "voir" (ou "entendre") dans une certaine zone autour d'eux, si le joueur est à proximité, ils se déplacent vers lui, si le joueur est adjacent, ils l'attaquent, si le joueur est loin, ils se déplacent aléatoirement. On pourra moduler leur comportement (par exemple s'ils sont adjacent au joueur, ils peuvent, de manière non-déterministe, soit l'attaquer, soit fuir, soit ne rien faire, soit tourner autour de lui, en fonction de leur férocité, de la présence ou nom d'autres monstres à proximité, ou de la santé du joueur).

4.4.3 Clefs

Certaines portes peuvent être fermées à clef, il faut trouver une clef (un objet posé quelque part dans le labyrinthe) pour les ouvrir. On pourra étudier dans les invariants la "faisabilité" d'un niveau (en vérifiant que la clef pour une porte ne se situe pas de l'autre côté de la porte, par exemple).

4.4.4 Trésor

Chaque niveau contient un trésor accessible. Pour que le niveau soit "gagné", le joueur doit s'échapper du labyrinthe par la sortie en emportant le trésor.

4.4.5 Pièges

Certaines cases peuvent être piégées: par exemple des trappes qui font des dégats au joueur, des téléporteurs qui transporte le joueur à un autre endroit de la carte, des tunnels desquels sortent un flux continu de monstre. On pourra faire que certains pièges soient interactifs: on peut les désactiver, par exemple avec une action spéciale (pousser un levier) ou un objet spécial à trouver dans le labyrinthe.

4.4.6 Niveaux

Le jeu se compose d'une succession de niveaux: à chaque fois qu'un joueur finit un niveau, il est transporté à l'entrée du niveau suivant en conservant ses points de vie et les éventuels objets qu'il a ramassés. Le jeu est "gagné" quand tous les niveaux ont été finis.

4.5 Extensions: Tartes à la crème

Cette sous-section présente des extensions "classiques" auxquelles on peut s'attendre en jouant à un *dungeon crawler*.

4.5.1 Ennemis Variés

Il existe plusieurs types de monstres différents, qui varient non seulement dans leur statistiques (plus ou moins dangereux, plus ou moins résistants, plus ou moins rapides) mais aussi dans leur comportement: certains peuvent uniquement entendre le joueur (on simulera le son en comptant le nombre de porte/mur qu'il faut franchir pour aller du joueur au monstre quand le joueur se déplace), certains peuvent uniquement voir le joueur (on calculera de la même manière des "lignes de vue"), certains peuvent tirer des projectiles, certains exhibent des comportements évolués (par exemple prendre le joueur à revers ou bloquer une sortie).

On peut, bien sûr, imaginer des monstres invisibles, des fantômes qui franchissent les murs, des "gobelins voleurs de clef", ...

4.5.2 Equipement et Statistiques

Le joueur peut trouver, ramasser et utiliser des armes, armures et outils qu'il peut trouver dans le labyrinthe ou sur le corps des monstres décédés. Elles peuvent améliorer ses capacités de combat ou de déplacement.

4.5.3 Consommables

Le joueur peut trouver, ramasser et utiliser des potions et de la nourriture dans le labyrinthe. La nourriture lui permet de ne pas mourir de faim (il faut implémenter un système de "survie" qui ferait perdre progressivement de la santé à un joueur qui ne mangerait pas), les potions peuvent avoir des effets divers et variés.

4.5.4 Pouvoirs

Le joueur peut choisir au démarrage (ou acquérir pendant le jeu) des pouvoirs spéciaux, qui lui permettent d'améliorer ses capacités de combat (lancer des projectiles, repousser les monstres vers l'arrière) ou de déplacement (sauter par dessus des "fosses", se téléporter, crocheter des portes).

4.6 Extension: Génération de Labyrinthes

Le jeu dispose d'un module de génération aléatoire de labyrinthes sains. Un invariant complexe permet de s'assurer que chaque labyrinthe généré est jouable (et intéressant).

4.7 Extension: Brouillard de Guerre

L'interface du jeu n'affiche que les cases proches du joueurs et masque les autres. Les cases situées derrière un mur ou une porte ne sont pas visibles par le joueur.

On peut ajouter une gestion de la lumière, qui influe sur ce qui est visible ou non (torche utilisable par le joueur, lampes fixes dans le donjon).

4.8 Extension: Première Personne

Le jeu est représenté à la première personne, comme dans *Dungeon Master*. On doit tenir compte de l'orientation du joueur (la direction dans laquelle il regarde) et des monstres. Un "moteur graphique" basique doit être utilisé pour pouvoir représenter les 9 (par exemple) cases se situant devant le joueur (pas besoin d'outils de graphismes 3D, on s'inspirera du mode de représentation de Dungeon Master, en modifiant la taille des *sprites* des objets éloignés).

4.9 Extension: Continu

Le déplacement du joueur est "continu" (comme dans le TME6), c'est-à-dire qu'il n'y a plus de cases (mais il y a toujours des murs et des portes) et le jeu est en temps réel (cf. *Gauntlet*). Cette extension nécessite de réfléchir à comment stocker un "labyrinthe continu", à comment le représenter avec SDL, et à gérer des déplacement à granularité fine pour les monstres et le joueur.

4.10 Extension: Multijoueur

Plusieurs joueurs peuvent jouer simultanément dans le même niveau (par exemple en utilisant les touches de "parties" différentes d'un clavier AZERTY), et interagir entre eux.

5 Rapport

Le rapport de projet se compose de quatre parties:

- un **manuel** d'utilisation succint expliquant comment tester le jeu depuis le projet *stack*.
- une liste exhaustive des **propositions** (invariants/pré-conditions/post-conditions) implémentées dans le projet, classées par types / opérations et contenant une brève description de ce que vérifie la propriété.
- une description des **tests** implémentés par le `Spec.hs` du projet.
- un **rapport** proprement dit, qui décrit le projet, les extensions choisies et leur implémentation et qui, en plus, explicite les points importants de l'implémentation, en mettant en évidence du code, des propositions, des tests pertinents (ou des bugs mis découvertes grâce à la méthode de développement sûre).

De manière générale, il est fortement conseillé de consigner dans le rapport (sous la forme d'un paragraphe) toutes les difficultés rencontrées dans le développement du projet et tous les *points forts* (tests basés sur la propriété, utilisation d'algébriques, invariants de type complexe, ...) du projet afin de les mettre en valeur pour l'évaluation.

6 Barême Prévisionnel

Cette suggestion de barême n'est pas opposable, il s'agit d'un exemple possible pour la notation du projet, destiné à faire comprendre les priorités d'évaluation:

- 4 points pour **le rapport**: le rapport doit être clair, lisible, et contenir les informations nécessaires (manuel, propositions, tests). Deplus, le rapport doit présenter suffisamment de points intéressants du développement du projet.
- 4 points pour l'implémentation correcte du **jeu de base** (Section 3) en SDL2.
- 2 points pour **les propositions**: la rédaction des invariants/pre-conditions/post-conditions est systématique, pertinente, et commentée.
- 2 points pour **les tests**: les tests sont complets, correctement structurés, utilisent la puissance du *property based testing* et commentés.
- 1 point pour l'utilisation pertinente **du test basé sur la propriété**.
- 1 points pour l'utilisation pertinente **des algébriques**.
- 6 points pour **les extensions**: un nombre suffisant d'extensions ont été choisies, implémentées dans un style fonctionnel et (autant que possible) "haskellien", augmentées de propositions et de tests adéquats (on rappelle qu'une extension qui n'est pas accompagnée de inv/pre/post et de tests n'est pas évaluée). La pertinence, la difficulté et l'orginalité des extensions choisies sont prises en compte.