Université de Namur

2019 — 2020

Amsi
INFOB313 / IHDCB335

Sujet de Labo

Table des matières

1 Introduction

Ce document constitue la première partie d'un Labo commun entre le cours d'Analyse et Modélisation des Systèmes d'Information (INFOB313/IHDCB335) d'une part, et le cours de Syntaxe & Sémantique (INFOB314/IHDCB332) d'autre part.

L'idée est de montrer comment le développement effectif d'une solution à un problème met en action les acquis conceptuels et techniques de chacune des matières : l'analyse du problème et sa modélisation d'une part, et les techniques de compilation (analyse syntaxique et sémantique, génération de code) d'autre part.

1.1 Langages de Modélisation Dédiés

Un Langage de Modélisation Dédié (Lmd, ou *Domain-Specific Language* en anglais, terme que l'on utilisera dans le reste du document) se focalise sur un domaine restreint : il peut être purement informatique (comme la définition de bases de données), ou métier (comme la définition de polices d'assurances), contrastant ainsi avec l'usage traditionnel des langages de programmation classiques (comme Java ou C) qui permettent d'effectuer n'importe quel calcul (on dit alors qu'ils sont *Turing-complets*).

Un Dsl cherche à capturer les possibles variations d'un domaine afin d'en faciliter la définition ; souvent d'ailleurs, un Dsl va de pair avec une infrastructure plus conséquente : les artéfacts définis et/ou produits à partir du Dsl vont venir s'intégrer au sein d'une infrastructure logicielle plus conséquente. Par exemple, un opérateur de téléphone va utiliser un Dsl pour définir l'ensemble des forfaits qu'il propose à ses clients, lui permettant d'ajuster leur durée, leur spécificité (quand ont lieu les périodes de gratuité des appels et/ou Sms, etc.) et leur tarif ; ce Dsl va alors modifier le comportement d'une lourde infrastructure gérant, entre autres, l'ensemble des antennes, le système de facturation..., sans pour autant devoir réécrire ces parties de logiciel à chaque nouveau forfait.

L'avantage d'un Dsl réside dans cette spécificité : plus restreint, il permet d'automatiser de nombreuses tâches allant de la vérification de la consistance des modèles durant les premières étapes du cycle de développement, jusqu'à la génération complète d'un code fonctionnel, en passant par l'analyse approfondie et la remontée d'erreurs, favorisant leur capture au plus tôt dans le cycle de développement, contribuant à une réduction drastique des coûts liés à leur maintenance.

Un Dsl reste un langage au sens informatique du terme : il faut donc en définir ses syntaxes (i.e., les phrases du langage que l'on considère comme valides) et sa sémantique (i.e., le sens, l'interprétation que l'on attache à ces phrases).

La syntaxe *abstraite* constitute la représentation interne du langage, celle que l'ordinateur (et le programmeur) utilise afin d'opérer les calculs nécessaires au traitement du langage (en particulier, elle est parcourue exhaustivement pour produire le code final). La syntaxe *concrète* s'adresse aux utilisateurs pour manipuler le langage : elle peut être purement textuelle, mais peut aussi avoir une forme plus graphique (souvent hybride, car elle mélange du texte à des formes géométriques), comme nous le montrons dans ce document. Les deux formes ne sont pas exclusives, elles dépendent du public auquel le langage s'adresse (typiquement, une syntaxe purement textuelle est traditionnellement réservée aux informaticiens).

L'opération permettant d'inférer la syntaxe abstraite à partir de la syntaxe concrète s'appelle le *parsing* : il est généralement plus facile pour des formes textuelles, et son étude est l'objet des cours de Syntaxe & Sémantique.

L'objectif général dans le cours d'Amsi consiste à définir des langages de modélisation suffisamment précis pour pouvoir modéliser en détail l'exemple proposé. Le même exemple sera repris dans le cours de Syntaxe & Sémantique pour étudier la génération de code à partir d'une grammaire.

1.2 Objectifs & Organisation

Ce document décrit le jeu en ligne *Golden Quest* édité par la société Coding Park. Il s'agit d'un *World Game* simplifié et éducatif : le but du joueur est de guider son personnage vers un item de sortie, afin d'accéder au niveau supérieur, avec pour but ultime de terminer le jeu.

À la différence d'autres jeux de ce genre, *Golden Quest* est un jeu d'inspiration pédagogique : il permet d'apprendre les principes de base de la programmation impérative, et offre une introduction de haut niveau à la (méta-)modélisation. L'une des applications directes de cette approche pédagogique est la possibilité de modéliser soi-même des niveaux dont on définit l'aspect et la solution.

Prenez le temps de vous familiariser avec le jeu (une question utilise l'éditeur en ligne) :

https://www.codingpark.io

Dans ce Labo, nous nous intéressons à la modélisation (partielle) du jeu proposé par Coding Park. La modélisation se réalise suivant les deux phases suivantes :

Décrire les *Plateaux de Jeu* (§2) et ses composantes, où il s'agit de modéliser les éléments constituant les leçons et niveaux.

Définir un *Langage d'Action* (§3) permettant de traduire en instructions précises les étapes guidant le personnage vers la sortie.

La définition du Langage d'Action repose sur les principes des langages de programmation généralistes (*General-Purpose Programming Languages*, GPLs) comme Pascal.

Pour les deux parties (Plateau de Jeu & Langage d'Action), votre travail consiste en la production de quatre artéfacts :

1. Un **Diagramme de Classes** capturant les concepts présents dans chaque partie ;
2. Un ensemble d'**expressions OCL** énonçant les contraintes et contrats portant sur les diagrammes précédents ;
3. Un **Diagramme d'Objets** illustrant une utilisation des diagrammes et expressions sur chaque partie ;
4. Un **niveau de jeu** original permettant d'accompagner un apprenant sur une leçon particulière.

Les parties forment un tout, dont la cohérence est une grande part de la note résultant de votre travail : en particulier, des diagrammes d'objets qui ne seraient pas conformes aux diagrammes de classes correspondants seront sévèrement sanctionnés.

Pour vous aider, une série de Questions (§4) vous guident dans la réalisation du Labo : nous vous demandons de suivre les questions pour faciliter **notre** lecture.

2 Plateau de Jeu

Le jeu est constitué de *leçons* (ou *quêtes*) évolutives qui comportent des *niveaux* (ou *cartes*) suivant une progression déterminée : un niveau ne peut être joué que si son précédent a été réussi. Pour chaque niveau, l'élève (ou *joueur*) doit aider Cody, le Robot Pirate, à trouver les trésors cachés dans les îles de son archipel. Pour ce faire, le joueur doit écrire le code, exprimé dans un Langage d'Action simplifié (nommé « *Play* » dans la suite), définissant le parcours et les actions de Cody pour arriver au trésor.

Un niveau est terminé si le coffre a été déterré (avant l'écoulement du chronomètre s'il y a lieu). Si un niveau est perdu (en tombant, en mourant, ou en n'ayant pas déterré le coffre avant le temps imparti), il est possible de le recommencer, ou d'éventuellement retourner à l'un des niveaux précédents de la même leçon.

2.1 Leçons & Niveaux

Chaque leçon du jeu aborde une thématique de programmation particulière, reprise dans son intitulé. Une leçon est organisée comme une quête : pour la réussir, il faut réussir chacun des niveaux qui la constituent. Un niveau est représenté visuellement par une carte sur laquelle Cody évolue. Leçons (cf. Figure 1a) et niveaux (cf. Figure 1b) sont organisés de manière séquentielle pour marquer une progression dans l'apprentissage.

Un niveau se construit à l'aide d'un *Éditeur de Niveau*, comme montré en Figure 2b : à partir d'un *Canvas* vide, ou d'un niveau déjà existant, une palette d'éléments graphiques permet de modifier le canvas afin de concevoir graphiquement les détails du niveau. Les dimensions de la carte d'un niveau peuvent être modifiées en jouant sur les triangles comportant des signes '+' et '-' à gauche et en bas, ce qui rajoute des cases vides (ou en enlève). L'espace ainsi défini peut être rempli par des éléments graphiques :

Élement de surface remplit une case par un pavé franchissable représentant un pont (franchissable), ou la texture plane du thème (gazon, glace ou roc) ;

Coffre remplit une case par un pavé contenant le coffre à découvrir dans le niveau ;

Obstacle dispose sur un élément de surface un obstacle infranchissable dépendant du thème (buisson, sapin enneigé ou pylône rocheux).

Téléportation dispose sur un élément de surface un élément de téléportation : soit un tunnel permettant le déplacement instantané vers l'autre tunnel de la même couleur ; soit un levier d'activation associé à une couleur, permettant d'activer les tunnels présents sur le (même) plateau et de la même couleur.

Personnage place sur un élément de surface un personnage.

Trois thèmes (plaine, neige, désert) sélectionnés par le joueur au début, fixent le type d'éléments de surface pour obtenir un rendu harmonieux. De nombreux niveaux pré-construits par les concepteurs du jeu sont fournis, mais un joueur peut concevoir un nombre arbitraire de niveaux, stockés dans son *Profil Utilisateur*. Ces niveaux pourront être mis à la disposition du public et échangés afin de favoriser l'apprentissage.

2.2 Personnage

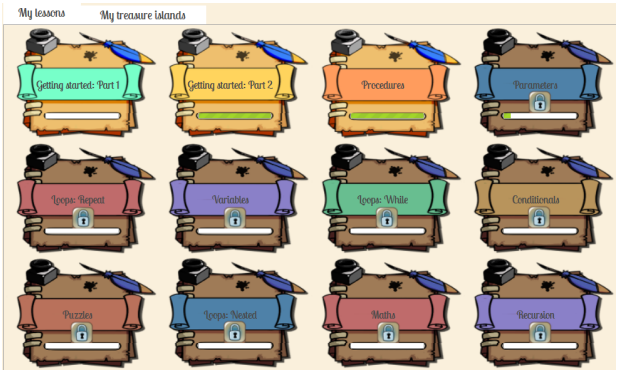
À partir de l'*Éditeur de Niveaux*, deux types de personnages peuvent être placés sur une carte, marquant leur position initiale (cf. Figure 2a) : Cody le Robot, le personnage représentant le joueur ; et des squelettes, qui peuvent réagir ou non au passage de Cody en l'attaquant, ou en restant parfaitement calmes. Le choix du placement des squelettes doit cependant laisser la possibilité d'atteindre le trésor.

En ouvrant un compte, un joueur dispose d'un *Profil Utilisateur* qui stocke des informations de compte (tel que l'email de connection, ainsi que d'autres informations à chercher vous-mêmes), mais aussi des informations sur l'évolution du joueur dans le jeu :

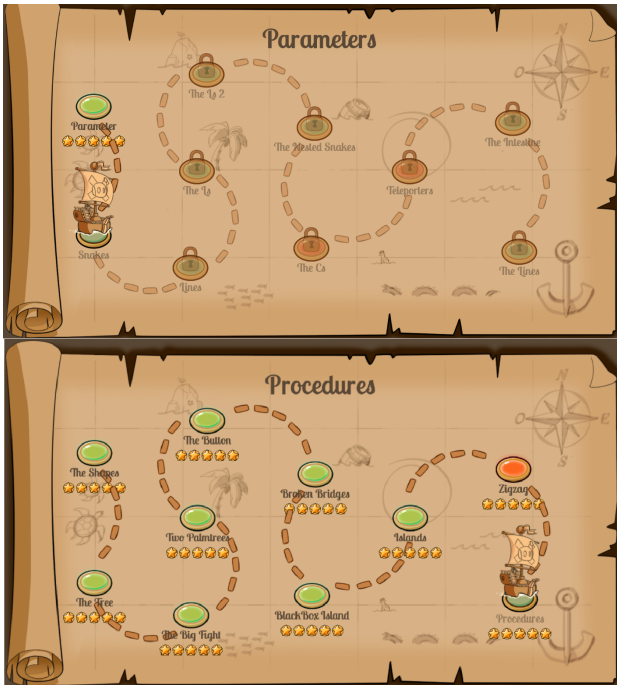
Niveau Maximal Atteint À tout moment, le *Profil* retient quel est le niveau le plus avancé que le joueur doit résoudre pour progresser ;

Performances Pour chaque niveau, plusieurs indicateurs de performances sont retenus :

- **Respect du Temps** indique si le niveau a été terminé avant l'écoulement du temps imparti, s'il y a lieu ;
- **Nombre de tentatives** indique combien de fois le code a été exécuté avant résolution du niveau ;
- **Distance du code** fournit une métrique estimant la différence entre un code idéal fourni par le concepteur, et celui proposé par le joueur.

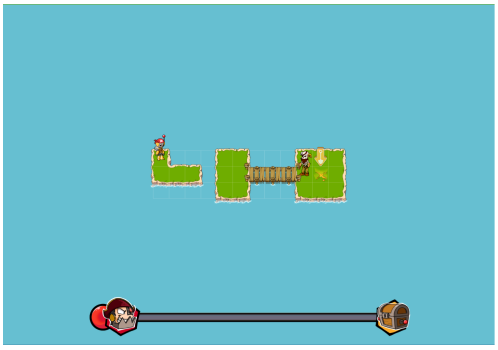


(a) Liste des Leçons (disponible à partir de la vue *Accueil*). Les trois premières leçons ont été complétées ; la leçon *Parameters* est en cours ; les suivantes sont encore verrouillées.



(b) Progression pour les niveaux *Parameters* et *Procedures*. La leçon sur les procédures a été complétée, avec un seul niveau qui a été recommencé (intitulé *Zigzag*, représenté en orange). La leçon *Parameters* est la leçon en cours, avec le premier niveau déjà réalisé et le deuxième en cours.

FIGURE 1 – Pédagogie organisée en leçons et niveaux.



(a) Canvas du Niveau *The Skeleton* de la leçon *Getting Started #1*. Cody, le personnage du joueur, se trouve sur la gauche ; un pirate gentil (c'est-à-dire, qui n'attaque pas) se trouve derrière le pont, empêchant l'accès au trésor. Le canvas de la carte est affiché avec la grille visible, et on voit, tout en bas, le compteur limitant le temps pour écrire une solution correcte.

(b) Editeur de niveau. À partir du niveau intitulé *The Skeleton* (cf. la version originale en Figure 2a), l'utilisateur a introduit un chemin constitué d'éléments de surface (gazon, puis pont) permettant de contourner le squelette, et ainsi éviter de se battre. La palette de conception sur le bord droit permet d'ajouter/d'effacer des éléments graphiques ; les flèches sur le canvas permettent d'augmenter la taille de la carte.

FIGURE 2 – Approche multi-vue : le *Canvas* de jeu peut recevoir en vue additionnelle la *Palette d'Édition*.

3 *Play* : Un Langage d'Action (étendu) pour guider Cody

Le Langage d'Action *Play*, proposé dans le jeu (basé sur la technologie DslForge [1]), est limité en termes d'expressivité. Par exemple, il ne permet de manipuler que des entiers : imaginez, alors, encoder l'équivalent des chaînes de caractères sur la base d'entiers uniquement !

Cette section propose de reprendre *Play* et de l'étendre pour améliorer l'expressivité du Langage d'Action afin de le rendre à la fois plus complet et plus convivial, en introduisant davantage de types primitifs, des structures de données plus riches, et la possibilité de stocker des informations afin de les réutiliser à un stade plus avancé dans le jeu.

3.1 Action / Programme & Déclarations

Un *Programme Play* définit les actions que Cody doit réaliser pour terminer un niveau. *Play* est constitué de déclarations qui peuvent être :

- soit une déclaration de procédure, qui comporte un nom et un corps, et déclare elle-même une liste de paramètres, et possède un type de retour ;
- soit une déclaration de variable, qui consiste en un nom associé à un type.

Schématiquement donc, une déclaration associe (*déclare*) un nom associé à un type choisi parmi ceux définis en Section 3.2. Seules les procédures peuvent être déclarées avec le type **void**.

3.2 Types

En plus du type entier déjà disponible, trois nouveaux types primitifs sont introduits : les booléens, les réels et les chaînes de caractères. Les notations pour les littéraux correspondant à ces types sont données dans le Tableau **??**. On introduira le type artificiel [2] **void** à utiliser pour les procédures (cf. Section 3.1).

En plus des types primitifs, deux types structurés sont introduits pour permettre une meilleure structuration et manipulation des données utilisateur. La syntaxe de ces types s'inspire directement de Pascal :

Énumerations définit une liste ordonnée des littéraux possibles constituant une énumération. Par exemple, le type WeekDay peut être défini par la déclaration suivante :

```
type WeekDay = { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
var day1, day2 : WeekDay;
day1 := WeekDay.Mon;
day1 := day2;
```

Pour éviter les ambigüités, on préfixera toujours les littéraux par leur énumération, comme montré pour la première affectation de day1.

Tableau définit une séquence de valeurs de même type, de longueur finie et prédéterminée. Un même tableau peut être déclaré de différentes manières, mais sera manipulé de manière uniforme. Par exemple, le code suivant définit une Matrix carrée d'entiers de trois manières distinctes :

```
type Matrix1 = array [1 .. 10] of array [1 .. 10] of integer;

type Line = array [1 .. 10] of integer;
type Matrix2 = array [1 .. 10] of Line;

type Matrix3 = array [1 .. 10, 1 .. 10] of integer;

var myMatrix, mm : Matrix1;
myMatrix[0,0] := 0;
myMatrix[0] := myMatrix[1];
myMatrix := mm;
```

1. https://dslforge.org/
2. Le type **void** est artificiel dans le sens où on ne peut pas déclarer une variable de ce type, mais simplement le faire apparaître dans la déclaration d'une procédure.

Les trois dernières lignes montrent des affectations valables pour les trois types de Matrix : la première ligne réalise l'affectation d'une case unique ; la deuxième ligne copie la ligne 1 dans la ligne 0 ; la troisième copie un tableau dans son intégralité.

Notons que les différents types d'accesseurs de tableau, çàd. les parties notées entre crochets (par exemple, [0,0] en ligne 6, ou [0] et [1] en ligne 7), ne peuvent s'utiliser qu'après une variable de type tableau. De même, les éléments accédés doivent être cohérents : en Ligne 6, l'accès à une cellule d'un tableau d'entier est utilisé au sein d'une affectation dont la valeur est entière ; en Ligne 7, l'accès à une colonne d'un tableau de tableaux d'entiers est utilisé dans une affectation avec une structure de même type.

Enregistrement définit un tuple de valeurs accessibles au travers d'un champ nommé à l'aide de la notation pointée. Par exemple, le code suivant définit un (type) enregistrement Complex, puis affecte la valeur 0 aux parties réelles et imaginaires d'une variable origin.

```
type Complex : record
    re : integer ;
    im : integer ;
endrecord
var origin : Complex
...
origin.im = 0 ;
origin.re = 0 ;
```

3.3 Instructions

Les instructions constituent le cœur de la définition des procédures : elles définissent les comportements possibles de Cody le Robot afin de le mener vers le trésor, tout en évitant les pièges parsemés dans le niveau (les pirates qui peuvent l'attaquer, et les trous qui peuvent le faire tomber).

Les instructions utilisent des expressions (cf. Section 3.4) pour calculer des valeurs et définir des tests, et peuvent être classées en différentes catégories :

Instructions Composées représentent des instructions qui sont composées d'autres instructions (y compris elles-mêmes). Elles sont de deux types :

Conditionnelle (if(...)) est une instruction spécifiant un choix conditionné par une garde booléenne exprimée par une expression ; une Conditionnelle se décline en deux versions, dont l'une permet de ne réaliser aucune action si la garde est fausse.

Itération est un type d'instruction spécifiant un calcul itératif / répétitif à l'aide d'une boucle while(...){...} ou do(...){...}. Structurellement, une itération possède toujours un corps (délimité par des accolades comme spécifié précédemment) composé lui-même d'instructions qui sont exécutées tant que la garde de l'Itération reste vraie.

Appel de procédure représente l'invocation d'une procédure définie dans le même programme, avec éventuellement une liste de paramètres effectifs. Les **instructions d'action** (comme left(), jump(), et leurs versions avec paramètres comme left(3)) sont traitées comme des appels de procédures singulières disponibles par défaut dans chaque programme *Play*, où l'appel sans paramètre (comme left()) correspond à un appel avec le paramètre par défaut 1 (çàd. traduit en left(1)).

Affectation permet d'assigner une valeur à un réceptacle, et se compose donc structurellement de deux parties :

- la partie gauche (*left-hand side*), apparaissant traditionnellement à gauche d'un symbole d"affectation (:= en Pascal ou simplement = en Java, on utilisera dans la Table 1 le symbole ⇐) désigne un réceptacle : soit une variable, soit une case, une colonne d'un tableau ou encore un tableau complet, ou bien encore un champ d'enregistrement ;
- la partie droite (*right-hand side*) est une expression permettant de calculer la valeur à affecter à cette partie gauche.

Des exemples d'expressions des différents types sont donnés dans la Table 1

3.4 Expressions

Les expressions permettent de calculer des valeurs qui apparaissent essentiellement dans trois endroits : les paramètres effectifs lors d'un appel de procédure ; les gardes des instructions composées, et les parties droites d'affectation.

Structurellement, les expression appartiennent à cinq catégories, dont des exemples sont données dans la Table 1 :

Litéral recouvre l'ensemble des expressions litérales, càd. dont l'interprétation correspond à elle-même.

Invocation désigne l'invocation d'une déclaration affectable, çàd. une variable, une cellule ou une partie de tableau, ou un champ d'enregistrement. Une invocation peut apparaître à droite comme à gauche d'une affectation.

Expression Binaire / Unaire désigne l'ensemble des expressions composées à partir d'un opérateur binaire (resp. unaire), à de deux (resp. d'une) sous-expression(s). Notons que n'importe quelle combinaison n'est pas possible, et il faudra s'assurer de construire des expressions bien typées.

Expression Parenthésée désigne une expression de groupage (typiquement, à l'aide de parenthèses, ou d'un symbole d'encapsulation graphique) constitué d'une sous-expression.

3.5 Tables Récapitulatives

La Table suivante récapitule les différentes constructions du Langage d'Actions *Play* en donnant quelques exemples significatifs pour chaque type d'instruction et d'expression.

	Catégorie	Type	Exemples	Sous-Expressions
Expressions	**Literal**	Entier	0 1	
		Booléen	true false	
		String	"super" "abcd"	
		Valeur d'Enumération	WeekDay.Mon	
	Invocation	Variable	v	
		Cellule	T[0, 0] T[0] T	
		Champ	origin.im origin	
	Unaire	Réel	-1.2	
		Boolean	**not** v	v
	Binaire	Numérique	v > 1 v = 1	v
		Boolean	a **or** b	a, b
	Parenthésée	Booléen	(**not** a) or b	a, b

	Catégorie	De	Exemples
Instructions	**Appel**	Action	jump() left(2)
		Général	rec() contourne(v+3)
	Affectation	Variable	myTab ⇐ T[1, v-1] myDay ⇐ WeekDay.Mon rotation ⇐ cplx.im + cplx.re
		Tableau	T[1, 1] ⇐ 1 T[n+1, m-1] ⇐ WeekDay.Mon M[1] ⇐ N[2]
		Champ	origin.im ⇐ 0 cplx ⇐ origin

TABLE 1 – Illustration d'expressions en syntaxe textuelle.

4 Questions

Le but de ce Labo est de modéliser le jeu tel qu'il est présenté dans les sections précédentes. La liste des questions de cette section vise à vous aider à obtenir des modèles fonctionnels en détaillant les étapes nécessaires à leur obtention : on adopte une approche top/down, c'est-à-dire qu'on va d'abord s'intéresser à une compréhension abstraite et haut niveau d'un concept, avant d'en raffiner la compréhension dans les questions ultérieures.

Les questions sont groupées en deux sections : on modélise d'abord le DSL décrivant les éléments graphique de leçons (§2), puis le DSL définissant les actions des personnages (§3).

Répondez à TOUTES les questions !

4.1 Modélisation du Plateau de Jeu

Cette section s'intéresse au plateau de jeu représenté par les éléments graphiques disposés sur la droite de la page web du jeu, comme décrit en §2. Les déliverables de cette section sont les suivants :

1. un Diagramme de Classes UML capturant les éléments graphiques ;
2. un ensemble d'expressions OCL capturant les contraintes sur ces éléments ;
3. un Diagramme d'Objets UML illustrant votre Diagramme de Classes ;
4. un niveau original défini à l'aide de l'Éditeur de Niveaux.

▷ **Question 4.1** ◁
Établir une première version d'un diagramme de classes UML qui fixe les éléments principaux : le jeu est constitué d'une série de leçons découpées en niveaux.

▷ **Question 4.2** ◁
Enrichir cette première version avec les détails nécessaires concernant les joueurs et leurs profils, ainsi que les niveaux. On prêtera une attention particulière aux éléments UML suivants :

— la multiplicité des associations doit correspondre aux contraintes de l'énoncé ;
— le type des attributs choisis doit permettre l'expression des contraintes ;
— les associations portant des agrégations / compositions doivent être choisies avec soin ;
— si besoin, la nature des collections (*bag*, *set*, *sequence*, *ordered set*) doit être justifiée ;
— les éventuelles hiérarchies doivent être pleinement spécifiées (complétude et couverture).

Pensez à justifier vos choix par rapport à l'énoncé ou au comportement du jeu lorsque vous précisez des hypothèses sur l'énoncé, en fournissant des explications de vos choix claires et fondées.

▷ **Question 4.3** ◁
Spécifier les contraintes suivantes, soit à l'aide d'éléments structurels dans le diagramme, éventuellement complétées par des contraintes en OCL (ceci dépend de la manière dont le diagramme de classes est construit) :

1. Les coordonnées d'une case ne peuvent excéder les dimensions du niveau ;
2. Chaque niveau ne contient qu'un seul Cody, et qu'un seul coffre ;
3. Un personnage ne peut pas se trouver sur un obstacle (même après un déplacement) ;
4. Deux personnages ne peuvent pas partager la même case ;
5. Le coffre doit se trouver sur un élément de surface franchissable ;
6. Chaque niveau comporte soit une paire de tunnels de téléportation de même couleur, soit un tunnel unique d'une couleur mais qui est initialement fermé ;
7. Un levier de téléportation ne peut être présent que s'il existe des tunnels de la même couleur.
8. Un obstacle (buisson, sapin ou pylône rocailleux) doit obligatoirement être posé sur une surface franchissable.

FIGURE 3 – Niveau simple pour la définition d'un Diagramme d'Objet (Question 4.4).

▷ **Question 4.4** ◁
Pour la Figure 3,

1. Définir le niveau décrit à l'aide d'un Diagramme d'Objets UML, en cohérence avec le Diagramme de Classe obtenu au terme de la Question 4.2 ;
2. Quelle propriété du jeu n'est pas satisfaite par ce niveau ?
3. Est-il possible de définir une contrainte OCL qui permette de vérifier cette propriété ? Pourquoi ?

▷ **Question 4.5** ◁
Définir, à l'aide de l'Éditeur de Niveau, un niveau original permettant d'illustrer les concepts de boucles imbriquées, de portée de variables, et de récursivité.

4.2 Modélisation de *Play*

Cette section s'intéresse au langage d'actions *Play*, comme décrit en §**??**. Les délivrables de cette section sont les suivants :

1. un Diagramme de Classes UML capturant la syntaxe de *Play* (instructions + expressions) ;
2. un ensemble d'expressions OCL capturant les contraintes sur les instructions et expressions ;
3. un Diagramme d'Objets UML illustrant votre Diagramme de Classes ; le second vous demande d'être créatif.

▷ **Question 4.6** ◁
Établir une première version d'un diagramme de classe UML qui fixe les éléments principaux : un Program(me) *Play* est un ensemble de déclarations.

▷ **Question 4.7** ◁
Modéliser le concept de Declaration.

▷ **Question 4.8** ◁
Raffiner le concept de Type en intégrant les types primitives et les types structurés (tableaux et enregistrements), de manière à pouvoir déclarer tous les exemples donnés en §3.2.

▷ **Question 4.9** ◁
Raffiner le détail du concept Instruction, introduit précédemment, conformément à la description en §3.3, de manière à pouvoir décrire tous les types d'instructions présentés.

▷ **Question 4.10** ◁
Raffiner le concept Expression, introduit précédemment, de manière à pouvoir décrire tous les types d'expressions décrits en §3.4, **sans exception et dans le détail**.

▷ **Question 4.11** ◁
Spécifier une contrainte OCL vérifiant l'unicité des déclarations au sein de leur contexte :

1. Les noms de procédures au sein d'une instance de programme ;
2. La procédure Cody au sein d'une instance de Program(me) ;
3. Les noms de variables (globales) au sein d'une instance de programme ;
4. Les noms des paramètres au sein d'une déclaration de procédure ;
5. Les noms de variables au sein du corps d'une déclaration de procédure ;
6. Les noms des champs au sein d'une déclaration d'enregistrement.

▷ **Question 4.12** ◁
Spécifier une contrainte OCL permettant de vérifier qu'une déclaration de type est bien formée :

1. la liste de champs d'un enregistrement est non-vide ;
2. un tableau comporte au moins une dimension qui doit être strictement positive.

▷ **Question 4.13** ◁
Spécifier en OCL le contrat OCL sur une opération type(exp : Expression) : Type qui renvoie le type d'une expression :

— Le type des littéraux est le type qui leur correspond (par exemple, true a pour type Booleen, 1 a pour type Entier, "aa" a pour type String) ;
— Le type d'une expression unaire est lié au type de son opérateur, à condition que sa sous-expression corresponde (par exemple, -1 doit avoir une sous-expression de type entier ou réel, et not b impose que b soit de type booléen) ;
— Le type d'une expression binaire est lié au type de son opérateur (similaire au cas unaire, à vous de trouver des exemples pertinents) ;
— Le type d'une expression parenthésée est le type de sa sous-expression ;
— Le type d'un accès à une variable est son type de déclaration ;
— Le type d'une expression gauche correspondant à l'accès à un champ est le type de sa déclaration dans l'enregistrement ;
— Le type d'une expression gauche d'accès à une case de tableau est le type de déclaration du tableau.

▷ **Question 4.14** ◁
Spécifier le contrat OCL sur une opération estValide() : boolean qui vérifie qu'une instruction est valide :

— Un appel de procédure doit référer à une déclaration de procédure dont le nom existe dans le programme.
— Les gardes des instructions composées doivent posséder un type booleen ;
— Les paramètres des instructions d'actions (right(), jump(), etc.) doivent être entier ;
— Le paramètre effectif de rang i dans un appel de procédure doit posséder le même type que le paramètre de même rang dans la déclaration du même nom que l'appel ;
— Les parties gauche et droite d'une affectation doivent être de même type ;
— Le type de retour d'une procédure doit toujours être **void**.

On utilisera l'opération type(exp : Expression) : Type définie à la Question précédente.

▷ **Question 4.15** ◁

Les instructions d'actions primitives de déplacement obéissent à une logique particulière en présence de certains éléments. En supposant l'existence d'une opération prec_mouv() : Déplacement qui retourne la direction du dernier déplacement effectué, spécifier les contrats OCL sur l'ensemble de ces instructions :

— Lorsqu'une telle instruction tente d'accéder une case où se trouve un obstacle, le déplacement n'est pas effectué ;

— Lorsqu'une telle instruction tente d'accéder une case où se trouve un tunnel, on ressort dans la case suivant le dernier mouvement à partir de l'autre tunnel ;

— Lorsqu'on saute dans une direction à partir d'une case, on atterit deux cases plus loin dans la même direction ; s'il y a un obstacle deux cases plus loin, on reste sur place.

▷ **Question 4.16** ◁

Pour la Figure 3,

1. Donner le code *Play* permettant de terminer le niveau, lorsque le sapin a été retiré du niveau ;
2. Donner le Diagramme d'Objet UML correspondant à ce code.

▷ **Question 4.17** ◁

Donner le code *Play* permettant de résoudre votre niveau original défini dans la Question 4.5.