Théorie des Langages : Syntaxe et Sémantique

Spécification du Langage: $PILS$

A Simple Language for Imperative Programming

Créateur : James Ortiz

Les développeurs savent pourquoi!

"Jamais la programmation n'a été aussi populaire"

Faculté d'informatique
Université de Namur

Année académique 2020 - 2021

Table des matières

1 Introduction

Ce document constitue la seconde partie d'un laboratoire commun entre le cours d'Analyse et Modélisation des Systèmes d'Information INFOB313/IHDCB335 d'une part, et le cours de Théorie des Langages : 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. Ce document constitue le document de spécification du DSL textuel, utilisé pour décrire le comportement d'un joueur dans un jeu pour apprendre la programmation.

La programmation est une activité pour le moins complexe et, quel que soit son âge, l'entrée en matière peut être à la fois impressionnante et déstabilisante. Se lancer dans l'apprentissage par le jeu est un excellent moyen d'assimiler des bases de manière simple et efficace, apprendre à connaître les bases d'un langage de programmation. Le jeu est un excellent moyen d'introduire des connaissances et de les approfondir grâce à l'expérience et la mise en contexte. Sans avoir la prétention de remplacer des heures ou des années de formation. Il y a des jeux (sur Internet) qui peuvent constituer une première étape pour forger ses premières armes. Par example :

- **CheckIO** : Sur CheckIO (https ://checkio.org/), vous avez le choix entre l'apprentissage de JavaScript et Python, le tout dans une interface innovante et agréable. La plateforme assigne le développeur à des missions qui donneront une série de point. Une fois accumulés, ces points donnent accès au niveau supérieur et d'autres missions.
- **Pixel** : Pixel est un projet avec lequel les enfants peuvent entrer en contact avec ces technologies de manière ludique sur téléphone et tablette. Les enfants peuvent ainsi programmer la personnalité et les fonctionnalités d'un robot, tout en se formant sur Scratch.
- textbfScreep : Screeps (https ://screeps.com/) est un jeu de stratégie en open source dédié aux programmeurs. La mécanique principale consiste en la programmation de l'IA des unités disponibles. Les joueurs contrôlent leur colonie dans cet univers persistant à l'aide du langage JavaScript.
- **CodeCombat** : CodeCombat (https ://codecombat.com/) est une plateforme de programmation qui s'adresse à un public large d'étudiants, de l'école primaire à l'université. Les étudiants de CodeCombat apprennent à écrire et structurer leur programme tout en jouant, leur permettant ainsi d'avoir un retour direct de leur travail.
- **Coding Park** : Coding Park (https ://codingpark.io/) 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. 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.

1.1 Un peu de contexte

L'apprentissage du code devient un élément nécessaire dans l'éducation des enfants. En effet, les générations futures auront à savoir manipuler ce type de langage pour développer des applications ou au moins connaître la gestion d'un logiciel qui fonctionne mal. En conséquece, de plus en plus de ressources existent afin de stimuler cet art de la programmation : **Coding Park**, **CheckIO**, **CodeCombat**. L'idée de ces jeux est de mélanger code et amusement. Dans chacune ces ressources, il faut programmer les déplacements d'un petit robot ou d'un personnage qui veut trouver des objets. Il faut s'assurer de lui dicter les commandes afin qu'il se déplace dans la bonne direction, saute au bon moment, presse des interrupteurs, etc. Le tout dans une progression qui permet de maîtriser peu à peu les différentes concepts du langages de programmation défini (un exemple est donné à la Figure 2).

En raison du caractère payant de toutes ces ressources, la faculté d'informatique de l'université de Namur a décidé de développer son propre langage de programmation pour aprendre à coder. Donc, la faculté d'informatique (je veux dire James Ortiz) a défini un DSL (le langage *PILS*) permettant de définir le comportement, mouvements et actions d'un personnage (ou avatar). Votre mission, si vous l'acceptez [1], est

1. De toute façon, vous n'avez pas le choix. Le projet compilateur fait partie intégrante de l'évaluation du cours

FIGURE 1 – La grille de jeu

d'écrire un compilateur transformant les programmes écrits en *PILS* dans un formalisme de bas niveau, exécutable par des robots (LEGO NXT http ://www.nxtprograms.com/). Le DSL permet de décrire le comportement du personnage sous forme d'une suite de conditions/actions sur base de leur environnement. Concrètement, le comportement du personnage est décrit dans un programme *PILS*. Le but avec notre langage de programmation (*PILS*) est d'apprendre la programmation à l'aide d'un langage simplifié (projet de cours d'Analyse et Modélisation des Systèmes d'Information (INFOB313/IHDCB335)). Dans le programme il y a une séquence d'instructions (mouvements) que le personnage doit éxecuter. Le présent document décrit la syntaxe et la sémantique du DSL *PILS*.

1.2 Syntaxe & Sémantique : de quoi s'agit-il ?

Comme expliqué au cours, spécifier la *syntaxe* d'un langage, revient à décrire l'ensemble des suites de symboles de l'alphabet qui sont des mots valides du langage. Il est très important de connaître cette syntaxe lorsque l'on utilise des langages de programmation, car elle permet de décrire des programmes qui vont pouvoir être lus et traités par le compilateur. Pour décrire de manière finie ces suites valides de mots du langage (dont le nombre est généralement infini, car représentant l'ensemble de programmes qu'il est possible d'écrire pour un langage), nous utiliserons une grammaire non-contextuelle. Celle-ci est donnée tout au long du document au format BNF (*Backus-Naur Form*) décrit un peu plus loin. Cette grammaire s'accompagne d'un ensemble de règles de bonne formation spécifiant, par exemple, le typage correct des expressions.

Une suite de symboles valides pour un langage n'a pas de signification en elle même. Le but de la *définition sémantique* est de leur fournir cette signification. Pour cela, on procède en deux étapes : d'abord, on fournit un *domaine sémantique*, qui est l'ensemble des significations possibles ; ensuite, on donne une fonction qui fait correspondre, à chaque élément syntaxique du langage, un élément du domaine sémantique. Par exemple, si l'on désire définir des nombres naturels dans un certain langage, l'on peut définir la syntaxe à l'aide d'une expression régulière $(0|[1-9][0-9]^*)$ et utiliser comme domaine sémantique l'ensemble $\mathbb{N}$. La correspondance entre une expression entière telle qu'on l'a définie (par exemple, 42) et un élément de $\mathbb{N}$ (qui sera donc dans notre exemple 42) est directe.

Le langage décrit dans ce document de spécification est de type un peu particulier, il s'agit d'un *DSL* (pour *Domain Specific Language*). Un DSL est un langage de programmation à l'expressivité limitée et se concentrant sur un domaine particulier (pour plus d'information, voir le cours d'Analyse et Modélisation de Systèmes d'Information, chapitre 6, Metamodelling). Par exemple, le HTML est un DSL utilisé pour décrire des documents hypertextes qui sera *interprété* par un navigateur ; un autre exemple est le langage utilisé par ANTLR pour décrire une grammaire et qui lui est *compilé* par celui-ci en un ensemble de fichiers Java permettant de traiter un texte respectant ladite grammaire.

IHDCB332/INFOB314.

1.3 Conventions de Notation

L'étude de cette Section n'est pas nécessaire en première lecture, mais servira de référence par la suite en donnant une explication détaillée, avec des exemples, des différentes notations utilisées dans le document.

1.3.1 Grammaire BNF

La forme de Backus-Naur (Bnf, *Backus-Naur Form*) est une notation couramment utilisée en informatique pour définir des langages de programmation et traduire des grammaires hors-contexte dans une forme lisible par un ordinateur. Cette notation est un *langage réflexif* : elle se décrit elle-même formellement par une grammaire, puisqu'elle est un langage permettant de décrire des langages. Au lieu de présenter une définition précise, nous introduisons le langage sur des exemples simples. On distingue trois types de symboles :

1. les *métasymboles*, i.e. les symboles propres à Bnf ;
2. les *non-terminaux*, des symboles définissant des catégories syntaxiques ;
3. et les *terminaux*, les symboles appartenant au langage décrit.

Par exemple, une instruction conditionnelle est une instruction possible dans un langage de programmation classique, ce qui pourrait être défini de la manière suivante :

```
     Statement   : :=  conditionalStmt | ...
ConditionalStmt  : :=  if expression then statement + (else statement *) ? done
```

Par convention, une définition Bnf sera écrite dans une police sans serif. Plusieurs choses sont à remarquer dans cet exemple :

- L'utilisation d'une *instance* d'un non-terminal s'écrit en minuscule et apparaît toujours à droite du signe : :=, alors que la définition d'un non-terminal s'écrit en majuscule, à gauche du : :=. Ainsi, on voit que la grammaire n'est pas complète : la définition de ConditionalStmt utilise une instance de Expression qui n'est définie nul part.
- Plusieurs terminaux appartenant au langage défini par la grammaire apparaissent ici en gras : **if**, **then**, **else** et **done**.
- Plusieurs métasymboles apparaissent dans cet exemple :
  - le symbole | dans la première ligne indique l'alternative : une Statement peut être une ConditionalStmt, mais aussi d'autres constructions non-précisées ici ;
  - les symboles + et * placés en exposants (par exemples statement$^+$ et statement*) indiquent qu'un élément (ici, statement) est répété respectivement au moins une fois, et zéro ou plusieurs fois ;
  - le point d'interrogation (?) après un (groupe de) symboles indique l'option : ici, la partie **else** est optionnelle et peut ne pas apparaître.

1.3.2 Sémantique formelle

Les éléments décrits par une grammaire Bnf n'ont pas de signification en tant que tel. Le but de la définition *sémantique* est de préciser cette signification. Pour cela, on procède en deux étapes. D'abord, on fournit un *domaine sémantique*, qui est l'ensemble des significations possibles. Ensuite, on donne une fonction qui fait correspondre, à chaque élément de $PILS$, un élément du domaine sémantique. Un exemple devrait éclairer cette obscure explication : prenons le langage des nombres $PILS$:

```
Entier  ::=  (-)? (Chiffre)+
Chiffre ::=  [0-9 ]
```

L'ensemble des nombres syntaxiquement valides est appelé Entier. Il nous faut d'abord trouver un domaine sémantique, c'est-à-dire un ensemble de valeurs que *représentent* les éléments de Entier. Nous choisissons $\mathbb{Z}$, l'ensemble des nombres entiers, positifs, négatifs ou nuls. Ensuite, nous devons expliquer comment interpréter les éléments syntaxiques. Nous devons donc faire correspondre, à chaque élément de Entier, un élément de $\mathbb{Z}$. Vu que Entier et $\mathbb{Z}$ sont infinis, nous ne pouvons pas, évidemment, écrire une longue table de correspondance, comme la table 1.

Précédemment, nous avons expliqué que l'on pouvait décrire les langages, qui sont des ensembles infinis, formellement et de manière finie, en utilisant des grammaires. Partant du même principe, nous

TABLE 1 – Une très longue table de correspondance

Entier	$\mathbb{Z}$
0	0
1	1
−1	−1
⋮	⋮
1578	1578
⋮	⋮

allons définir, formellement et de manière finie, les fonctions qui, à chaque élément du domaine syntaxique, font correspondre un élément du domaine sémantique. Pour obtenir cette définition finie, nous utiliserons typiquement une induction sur la structure des éléments syntaxiques.

La manière de procéder sera donc la suivante. Pour chacune des formes possibles des termes du domaine syntaxique, nous dirons comment l'élément du domaine sémantique peut être calculé. Le grand principe qui soutient la *sémantique dénotationnelle* est la *compositionnalité* : la sémantique d'un élément ne peut être définie qu'en fonction de la sémantique de ses composants directs. Par exemple, si l'on se réfère à Entier, l'ensemble des nombres $PILS$, nous avons décidé que le domaine sémantique était $\mathbb{Z}$. La fonction qui fera correspondre à chaque élément de Entier un élément de $\mathbb{Z}$ se notera : $[\![.]\!]$. Elle possède la signature suivante :

$$[\![.]\!] : \mathsf{Entier} \to \mathbb{Z}$$

La flèche $\to$ signifie que $[\![.]\!]$ est une fonction totale : elle est définie pour tous les éléments de Entier. Nous définissons $[\![.]\!]$ par les équations ci-dessous. Notez l'utilisation des doubles crochets $[\![$ et $]\!]$. Ils ne servent qu'à encadrer les éléments qui appartiennent au domaine syntaxique, de façon à mettre en évidence la distinction entre éléments *syntaxiques* et *sémantiques*.

$$\begin{array}{rcl} [\![-n]\!] & = & 0 - [\![n]\!] \\ [\![n \cdot c]\!] & = & (10 * [\![n]\!]) + [\![c]\!], \text{ avec } n \in \mathit{Chiffre}^+ \\ [\![\mathbf{0}]\!] & = & 0 \\ [\![\mathbf{1}]\!] & = & 1 \\ & \vdots & \\ [\![\mathbf{9}]\!] & = & 9 \end{array} \tag{1}$$

Si il s'agit d'une suite de chiffres n, précédés d'un signe $-$, alors, l'entier correspondant est l'opposé de l'entier représenté par n. Si n est un simple chiffre, alors, on donne, par une liste *finie*, l'entier qui lui correspond. Par exemple, **3** correspondra à 3. Si n peut être divisé en deux parties : n' et c telles que n' est une suite de chiffres (non vide) et c est un chiffre, alors, il faut "décaler vers la gauche" l'entier représenté par n' et y ajouter la valeur de c dans la partie des unités.

1.3.3 Sémantique formelle et spécification d'un compilateur

La description syntaxique du langage permet au programmeur ou au constructeur d'un compilateur, de déterminer, sans ambiguïté possible, quelles constructions sont légales dans le langage et lesquelles ne le sont pas. L'utilité de la définition sémantique peut sembler moins frappante. Pourtant, il suffit de réfléchir aux nombreux choix que vous pouvez poser si vous devez interpréter un appel de procédure. Comment sont passés les paramètres ? Que signifie *passage par adresse* ou *passage par résultat* ? Dans quel ordre dois-je évaluer les composantes d'une expression ? Avoir une sémantique formelle permet de répondre à ces questions, car elle n'induit qu'une seule interprétation possible des programmes écrits dans un langage. Donc, dans un monde idéal, les langages seraient tous dotés d'une sémantique formelle, que les programmeurs consulteraient avant de construire les compilateurs. De cette manière, un programme d'un langage $\mathcal{L}$ donnerait les mêmes résultats, qu'il soit compilé et exécuté sur une plate-forme X ou Y.

Votre travail, lors de ce projet, sera de construire un *compilateur*. Un compilateur est un programme qui, prenant en entrée un programme P_s, écrit dans le langage *source* Source, produit en sortie un programme P_t dans le langage *cible*, Target. Cette traduction doit *préserver la sémantique de* P_s. Formellement, étant donné un domaine sémantique $\mathbb{D}$ et deux fonctions sémantiques : $[\![\cdot]\!]_s : \mathit{Source} \to \mathbb{D}$ et

$[\![\cdot]\!]_t : Target \to \mathbb{D}$, si P_t est le programme résultant de la compilation de P_s, alors,

$$[\![P_s]\!]_s = [\![P_t]\!]_t$$

Le compilateur que vous devez construire pourra traduire n'importe quel programme $PILS$ syntaxiquement valide en un programme équivalent. De plus, votre compilateur devra refuser de traduire des textes qui ne sont pas des programmes $PILS$ syntaxiquement valides.

1.4 Organisation du Document

Ce document décrit complètement un DSL utilisé par une société fictive, **World Game fictif**, spécialisée dans la production de world games. À la différence du Labo en Amsi, le formalisme choisi pour décrire le DSL repose cette fois sur une représentation textuelle sous forme de grammaire, qui se prête davantage à l'étude des techniques de compilation.

L'objectif est ici de produire un compilateur fonctionnelle réaliste, implémenté en Java, à partir des deux phases suivantes :

1. Un fichier **décrivant les mondes** et ses composantes (avec l'extension .wld), où il s'agit de modéliser les éléments constituant le monde, le plateau de jeu ainsi que ses différents constituants, mais aussi les personnages (personnage principal, zombies, etc) qui y évolueront.
2. Un fichier **décrivant la stratégie** de jeu du personnage principal écrit en $PILS$ (avec l'extension .B314), produit le code machine (NBC (Next Byte Codes)) permettant d'installer et exécuter ce code dans des robots (LEGO NXT 2.0) (avec l'extension .nbc).

Le reste du document décrit les différents éléments du DSL. Chaque section suit un schéma descriptif identique : la *syntaxe* du langage concerné est présenté sous forme de grammaire ; la *sémantique*, c'est-à-dire les structures mathématiques associées sont détaillées en montrant le lien avec l'écriture sous forme de grammaire.

2 Description du monde

Un monde est représenté par un plateau de jeu carré de dimension fixe (**arena as square** [27, 27]), sur lequel les personnages peuvent se déplacer. La plupart des cases sont vides, mais certaines contiennent des obstacles (murs ou rochers, arbres, menhirs, etc), des items (nourriture, boissons, minitions, les bottes, kits de plongée, bonus, etc) ou des orientations (radar, carte). L'idée avec le langage $PILS$ est d'écrire les actions/mouvements pour aider les personnages à trouver le Graal et éviter les obstacles, zombies, etc. Le Graal, les obstacles, les items, les orientations (plateau de jeu) peuvent être décrits (en utilisant notre langage $PILS$) dans un fichier avec l'extension ".wld". Chaque fichier (".wld") contient une description du monde comme s'il s'agissait d'un puzzle. Toutes les obstacles, items, orientations dans les cases ont la memê taille. Chaque obstacles, items, orientations dans les cases ont son prope représentation dans le fichier ".wld". Le fichier ".wld" peut être créé ou ouvert avec un éditeur de texte (TextMate, Notepad, vi, etc). Il est possible d'ajouter des lignes de commentaires dans le fichier. Une telle ligne commence par le symbol // un commentaire d'une seule ligne ou des commentaires commençant par /∗ et finissant par ∗/. Ces commentaires peuvent s'étendre sur plusieurs lignes, mais ils ne peuvent pas être imbriqués. Les lignes de commentaires n'auront aucun effet sur le plateau de jeu. Donc, les cases du plateau de jeu, peuvent être remplies avec les éléments suivants :

— **Obstacles :** Les obstacles peuvent être infranchissable, comme le serait un rocher ou mur, impliquant qu'un joueur ne peut jamais accéder à cette case ou franchissable, lorsque ce sont des ronces ou des buissons, si le joueur parvient à les détruire. Les mondes sont aussi parsemés de zones de feu et d'eau. Rester sur une case de feu sans des bottes pare-feu retire un point de vie par tour, tandis que les zones d'eau ne peuvent être traversées sans avoir le kit de plongée dans son inventaire (le jeu empêche simplement le passage sur la zone).

— **Items :** Plusieurs items sont distribués dans les mondes pour aider les personnages dans leur quête : la nourriture et les boissons remontent le potentiel de vie du personnage qui les utilise ; les recharges de munition permettent de tirer à distance sur un ennemi ; les bottes pare-feu et les kits de plongée permettent de franchir les zones de feu et d'eau respectivement. Des bonus agrémentent certains mondes : ils permettent de faire évoluer les caractéristiques de combat d'un personnage en améliorant ses capacités d'attaque ou de défense.

— **Orientation :** Deux items sont d'une aide importante pour l'orientation et les décisions stratégiques : un radar et une carte permettent de déterminer respectivement la position de l'autre personnage sur le plateau, et celle du Graal. Cette position est signalée par une flèche donnant la direction à vol d'oiseau vers la cible : pour la carte, cette position reste affichée jusqu'à la fin de la rencontre ; pour le radar, cette position reste affichée durant n tours, nombre qui est doublé à chaque fois que le personnage ramasse un nouveau radar. Les personnages (qui représentant le joueur) peuvent se déplacer sur le plateau de jeu avec des *conditions/instructions*. Les zombies peuvent réagir ou non au passage du personnage principal en l'attaquant, ou en restant parfaitement calmes. Le choix du placement des zombies doit cependant laisser la possibilité d'atteindre le Graal. Une condition est une expression booléenne qui porte sur les variables locales (variables définies dans une fonction) ou les variables globales (variables définies au début du programme ou procédure) dont la valeur est conservée d'un tour de jeu. Les instructions, rédigées dans un langage impératif classique, permettent d'effectuer différent calculs aidant à la prise de décision. Une suite d'instruction devra (normalement) se terminer par la spécification de la prochaine action que le personnage doit effectuer (avancer dans une direction donnée, chercher le Graal, etc.).

Un exemple simple de la carte de jeu serait donc :

```
/* Comments are between delimiters */
declare and retain
        /* Global declarations */
        i as integer;
        j as integer;
        arena as square[27, 27];

by default do
        set i to 1
        set j to 1


while i < 17 do
  while j < 17 do
   set arena[i, j] to vines
   set j to j + 1
   done
  set i to i + 1
done


        set i to 0


while i < 18 do
  set arena[i, 0] to rock
  set arena[i, 17] to rock
  set i to i + 1
done


        set j to 1


while j < 18 do
  set arena[0, j] to rock
  set arena[17, j] to rock
  set j to j + 1
done


set arena[1,1] to player
```

```
set arena[22,14] to ennemi
set arena[8,12] to zombie
set arena[5,5] to zombie
set arena[2,7] to rock
set arena[3,7] to rock
set arena[4,7] to dirt
set arena[3,6] to rock
set arena[4,10] to fruits
set arena[3,12] to map
set arena[4,4] to radio
set arena[12,7] to soda
set arena[25,25] to graal
done
```

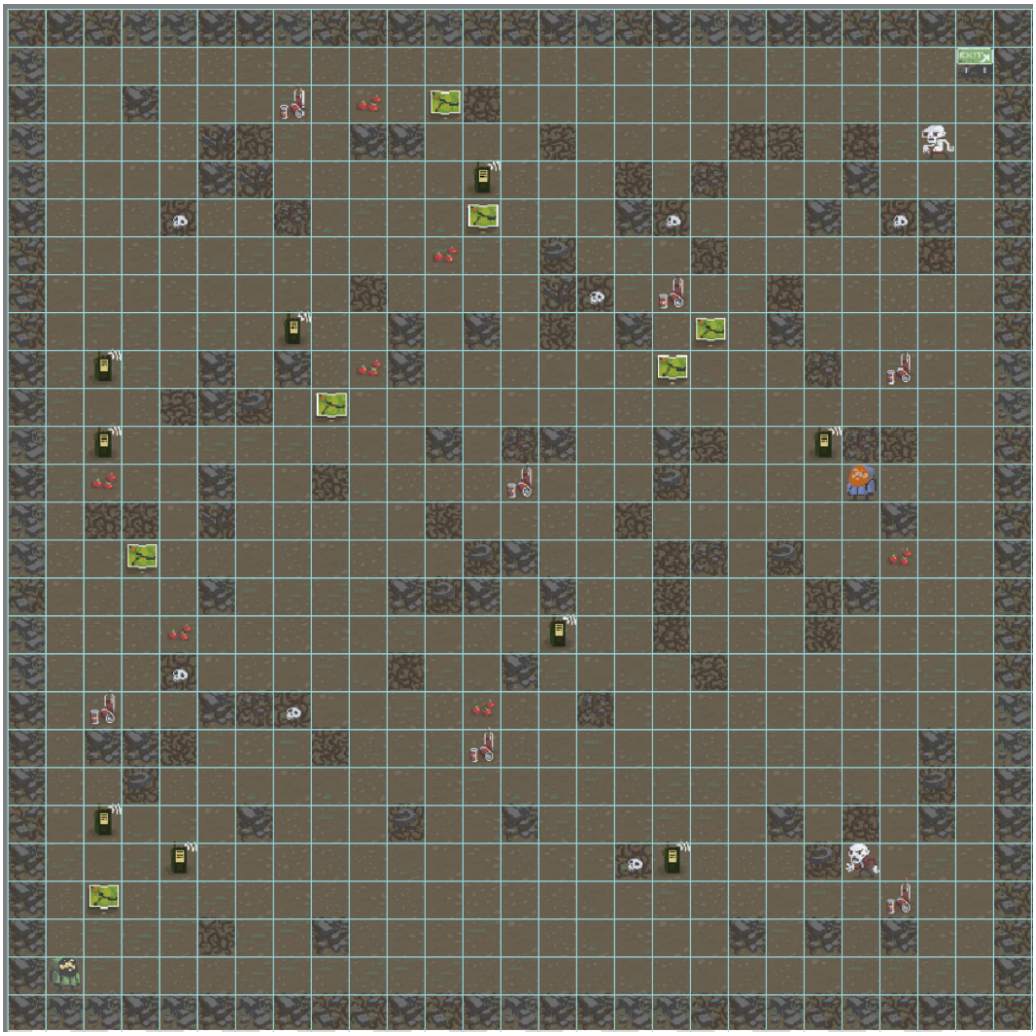

FIGURE 2 – Le plateau de jeu

Donc, La figure 2 représente graphiquement le fichier ".wld" défini ci-dessus (voir figure 2). La carte de la figure 2 donne un exemple de monde de taille 27 × 27. Le monde est bordé de murs (ou rochers) infranchissables (lignes et colonnes 0 et 26) empêchant les joueurs, placés ici en positions (1, 1) et en (22, 14) (on nommera le second "le roux" à cause de sa barbe rousse), de sortir autrement qu'en trouvant le Graal placé en (25, 25). D'autres obstacles parsèment le monde : par exemple, en (8, 2), et verticalement entre (24, 5) à (24, 7) se trouvent d'autres murs (ou rochers) infranchissables, tandis qu'en (4, 21) et (7, 21) se trouvent des obstacles qu'un joueur peut éliminer, le premier obstacle en (4, 21) avec la tête de mort nécessitant d'être frappé plusieurs fois avant de disparaître (et a donc une valeur d'item plus importante). Le premier joueur, en bas à gauche, est chanceux : il trouvera à proximité une carte en (2, 3)

et un radar en (2, 5) ou (4, 4). Le second joueur, lui, trouvera de la nourriture en (23, 12) et une boisson (23, 17). Il n'y a par contre aucun bonus ni aucune recharge de munition dans ce monde : les joueurs devront mener leurs combats au corps à corps, sauf s'ils ont accumulé des munitions dans les mondes précédents.

3 Description de la stratégie

L'idée avec le langage *PILS* est d'écrire les actions/mouvements pour aider les personnages à trouver le Graal et éviter les obstacles, zombies, etc. Le langage *PILS* capture la syntaxe du code permettant de définir les actions/mouvements du personnage, afin de naviguer au travers des différents niveaux et ainsi terminer le jeu. Un fichier *PILS* (avec l'extension ".B314") donne pour un personnage, sa stratégie de jeu sous forme de *condition/instructions*. Le fichier ".wld" peut être créé ou ouvert avec un éditeur de texte (TextMate, Notepad, vi, etc).

- **Programme et Fonctions** : Un Programme *PILS* définit les actions que le personnage doit réaliser pour terminer un niveau. La fonction définit le point d'entrée du programme, indiquant où commencer l'exécution des actions que le personnage doit entreprendre. Une fonxtion comporte un nom et un corps, et déclare une liste de paramètres.
- **Condition** : Une condition est une expression booléenne qui porte sur les variables d'environnement (par exemple, la position du joueur, le nombre de munition, *etc.*) ou les variables globales (variables définies par le programmeur *PILS*) dont la valeur est conservée d'un tour de jeu à l'autre.
- **Instructions** : Les instructions, rédigées dans un langage impératif classique, permettent d'effectuer différent calculs aidant à la prise de décision. Une suite d'instruction devra (normalement) se terminer par la spécification de la prochaine action que le personnage doit effectuer (avancer dans une direction donnée, utiliser une ration de nourriture, etc.).
- **Expressions** : Les expressions permettent de calculer des valeurs qui apparaissent dans les paramètres effectifs et dans les gardes des instructions. Des exemples simples de chaque catégorie d'expressions sont :
  - **Litéral** : Recouvre l'ensemble des expressions litérales(1, 1.2, true, false, "samedi", ...),
  - **Variable** : Désigne l'invocation d'un nom de variable (x, y, z, ...),
  - **Binaire** : Désigne l'ensemble des expressions composées à partir d'un opérateur binaire à de deux sous-expression(s) (x + 1, 1 > 2, ...),
  - **Unaire** : Désigne l'ensemble des expressions composées à partir d'un opérateur unaire à d'une sous-expression(s) (-1, not x, ...),
  - **Parenthésée** : Désigne une expression de groupage (typiquement, à l'aide de parenthèses, ou d'un symbole d'encapsulation graphique) constitué d'une sous-expression (x + 2 - y, ...).

Un exemple simple de stratégie écrite en *PILS* serait donc :

```
/* Comments are between delimiters */
declare and retain
import inputFile.wld

        /* Global variables */
        x as integer;
        b as boolean;
when your turn
        /* Conditions/instructions evaluated at each turn */
        when x > 5 and life < 2 do
                next use soda
        done
        by default do /* Default */
                set x to x + 1
                next move west
        done
```

Dans la suite de ce document, nous détaillerons les différentes briques du langage à assembler afin de définir le monde et la stratégie *PILS*.

4 Domaine sémantique

Une stratégie $PILS$ dialoguera avec son environnement grâce à des entrées/sorties sous forme d'entiers (dont les valeurs sont fixées par convention dans la suite de ce document). Par exemple, la notification du prochain mouvement à effectuer se fait via une valeur entière particulière, imprimée en sortie lors de l'exécution. Nous considèrerons donc qu'un programme $PILS$ (résultant de la compilation d'une stratégie $PILS$) est un transformateur de suites d'entiers. Il reçoit en entrée une suite infinie d'entiers et produit en sortie une suite d'entiers. Il se peut également qu'il produise une erreur.

Afin de transformer son entrée en la sortie désirée, le programme manipule deux structures de données : une mémoire et un environnement. La *mémoire* ($\sigma \in \mathbb{S}$) assigne des valeurs à des adresses. Certaines adresses ne sont pas utilisées. La mémoire sera donc modélisée par une fonction *partielle*. Afin de simplifier la présentation et d'abstraire les choix d'implémentation, nous supposerons que la mémoire est infinie. Une valeur sera toujours une valeur soit entière, soit booléenne. L'*environnement* ($\epsilon \in \mathbb{E}$) permet d'obtenir l'adresse correspondant à chaque identificateur de variable (Id) apparaissant dans le programme. Il permet aussi de retrouver la fonction associée à chaque identificateur de fonction ou de procédure. Il est important de se rendre compte qu'un identificateur peut être attaché à différentes adresses, en fonction du contexte dans lequel il est évalué. Un environnement sera donc une suite de fonctions, appelées contextes et retournant pour un identifiant donné, soit un couple $(type, adresse)$, soit une fonction :

$$contexte : \mathsf{Id} \nrightarrow (\mathbb{T} \times \mathbb{A}) \cup \mathbb{F}$$

Où $\mathbb{T}$ est l'ensemble des types possibles en $PILS$, $\mathbb{A}$ est l'ensemble des adresses possibles et $\mathbb{F}$ est l'ensemble des sémantiques des fonctions (en claire, "ce qui se passe" lorsque l'on appelle une fonction déclarée). Habituellement, lorsque l'on décrit des suites (ou des mots), on utilise la notation "$\cdot$" pour représenter la concaténation de deux séquences ; nous ne dérogerons pas à cette tradition. Soit un environnement $\epsilon = c_1 \cdot \ldots \cdot c_n$. e peut être vu comme une pile, où les contextes les plus récents sont "au-dessus" (position n) et les contextes les plus vieux "en-dessous" (position 1).

$$\epsilon = \begin{pmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \end{pmatrix}$$

Pour trouver l'élément auquel un identificateur Id correspond dans l'environnement ϵ, on regarde d'abord s'il est défini dans le contexte sur le dessus de la pile. Si Id n'est pas défini dans le contexte courant (celui au-dessus de la pile), on regarde s'il est défini dans un contexte antérieur.

$$\begin{array}{lcll} \epsilon(\mathsf{Id}) & = & c_n(\mathsf{Id}) & \text{si } c_n(\mathsf{Id}) \neq \bot \vee n = 1 \\ \epsilon(\mathsf{Id}) & = & (c_1 \cdot \ldots \cdot c_{n-1})(\mathsf{Id}) & \text{sinon} \end{array}$$

Où $\bot$ est la valeur retournée dans le cas d'un Id non défini dans le contexte courant. Par exemple, en supposant que l'on ait un environnement ϵ défini comme ci-dessous et que l'ensemble des identificateurs soit $\{x, y\}$

$$\epsilon = [x \mapsto (Int, 123)] \cdot [x \mapsto g, y \mapsto (Int, 256)] \cdot [y \mapsto (Bool, vrai)]$$

Alors $\epsilon(x) = g$ et $\epsilon(y) = vrai$.

4.1 Définitions

Formellement, afin de définir la sémantique de $PILS$, nous utiliserons les ensembles suivants :

- $\mathbb{Z}$ représente l'ensemble des entiers.
- $\mathbb{B}$ représente l'ensemble des booléens : $\mathbb{B} = \{vrai, faux\}$.
- $\mathbb{C}$ représente l'ensemble des types de cases que l'on peut trouver sur une grille de jeu : $\mathbb{C} = \{i | i \in \mathbb{Z} \wedge 0 \leqslant i \leqslant 10\}$. La correspondance entre un type de case donné (par exemple, **rock**, **player**, **zombie**, *etc.*) et un entier entre 0 et 10 est fixé par la sémantique associée aux terminaux **rock**, **player**, **zombie**, *etc.* décrite plus loin.
- $\mathbb{V}$ est l'ensemble des valeurs ($\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \mathbb{C}$) que peut prendre une variable en $PILS$.

$\mathbb{T}$ est l'ensemble des types possibles que l'on trouve en $PILS$. $\mathbb{T} = \{Int, Bool, Case, Tableau\}$, où $Tableau$ est une fonction retournant un élément de $\mathbb{T}$, ce qui permet de définir des tableaux à plusieurs dimensions.

$\mathbb{A}$ est l'ensemble des adresses de la mémoire. Nous posons $\mathbb{A} = [1..max]$ où max est supposé suffisamment grand pour permettre l'exécution de n'importe quel programme $PILS$.

$\mathbb{S}$ est l'ensemble des mémoires, c'est-à-dire des fonctions de signature $\mathbb{A} \nrightarrow \mathbb{V}$. Certaines adresses sont allouées mais non initialisées.

$\mathbb{I}$ **et** $\mathbb{O}$ sont les ensembles d'inputs et d'outputs, par définition, $\mathbb{I} = \mathbb{O} = \mathbb{Z}^*$,

$\mathbb{S}tate$ est l'ensemble des *états* du programme. Un état doit contenir toute l'information qui permet de calculer l'effet d'une instruction. Nous définissons $\mathbb{S}tate$ comme $(\mathbb{S} \times \mathbb{E} \times \mathbb{I} \times \mathbb{O})$. Nous noterons un état s_i donné comme un quadruplet $(\sigma_i, \epsilon_i, i_i, o_i)$, où σ_i représente la mémoire, ϵ_i représente l'environnement, i_i représente les inputs et o_i représente les outputs en l'état s_i.

$\mathbb{R}$ est l'ensemble des *résultats* de l'évaluation de fonctions. Lorsqu'une fonction est évaluée, elle modifie l'état courant (par effets de bords) et retourne une valeur. $\mathbb{R} = \mathbb{V} \times \mathbb{S}tate$. Pour l'évaluation d'une fonction, le résultat de cette évaluation sera donc un couple (v, s), où v est la valeur de retour et s est le nouvel état du programme.

$\mathbb{F}$ est l'ensemble qui représente les sémantiques des fonctions. Chaque fonction $PILS$ f correspondra à un élément de $\mathbb{F}$. Lorsque l'on invoque une fonction, l'état est modifié et une valeur peut être retournée. Le résultat de l'appel dépend de deux choses : l'état dans lequel la fonction est appelée et la valeur des paramètres effectifs. Il est possible que la fonction se plante, ce qui explique pourquoi f est partielle sur $\mathbb{R}$.

$$f \in \mathbb{F}, f : \mathbb{S}tate \rightarrow \mathbb{V}^* \nrightarrow \mathbb{R} \cup \mathbb{S}tate$$

$\mathbb{E}$ est l'ensemble des environnements. Dans $PILS$, les choses sont un peu particulières car dans un environnement donné, on peut trouver une variable et une fonction partageant le même identifiant. Les environnement sont en fait des piles de la forme [2] :

$$(\mathsf{Id} \times \mathbb{C}ategorie \nrightarrow \mathbb{F} \cup (\mathbb{T} \times \mathbb{A}))^+$$

Où $\mathbb{C}ategorie = \{vp, func\}$ et Id est l'ensemble des identifiants défini dans la section suivante.

5 Identificateurs

Les identificateurs sont utilisés pour identifier les variables et les fonctions définies en $PILS$. Un identificateur (Id) est un mot commençant par une lettre suivie de lettres et/ou de chiffres, comme spécifié dans la grammaire ci-dessous. Bien évidement, les mots clés du langage sont exclus de l'ensemble des identifiants possibles. Nous ne définissons pas ces mots-clefs en extension, mais le lecteur les découvrira en parcourant les différentes règles tout au long de ce document.

```
    Id   ::=  Lettre (Chiffre | Lettre)*
Lettre   ::=  [A-Za-z ]
```

Le domaine sémantique des identificateurs est un peu particulier. Il s'agit de l'ensemble (syntaxique) des identificateurs. Un identificateur sera interprété par lui-même. La fonction sémantique sera simplement la fonction identité : $[\![\mathsf{Id}]\!] = \mathsf{Id}$.

6 Déclarations de variables

En $PILS$, les variables peuvent se déclarer à deux niveaux : au niveau global les variables persistent en mémoire d'un tour de jeu à l'autre et ont une portée globale sur l'ensemble du programme (*i.e.*, elles sont accessibles à chaque endroit du code) ; au niveau local (au sein d'une fonction par exemple), les variables ont une portée limitée à la fonction (et n'existent donc que dans l'exécution de cette fonction).

2. en fait, c'est une pile d'éléments de l'union $(\mathsf{Id} \times \{vp\} \rightarrow (\mathbb{T} \times \mathbb{A})) \cup (\mathsf{Id} \times \{func\} \rightarrow \mathbb{F})$

6.1 Types de données

Commençons par donner les différents types existant en $PILS$, ainsi que leur sémantique, remarquez que la grammaire limite l'utilisation des tableaux aux tableaux à 2 dimensions :

Type	::=	Scalar | Array
Scalar	::=	**boolean** | **integer** | **square**
Array	::=	Scalar[(Chiffre)$^+$ (, (Chiffre)$^+$)?]

La sémantique d'un type ($[\![.]\!] : \text{Type} \rightarrow \mathbb{T}$) correspond à un élément de l'ensemble $\mathbb{T}$ et se définit de la manière suivante :

$$\begin{aligned} [\![\textbf{boolean}]\!] &= Bool \\ [\![\textbf{integer}]\!] &= Int \\ [\![\textbf{square}]\!] &= Case \\ [\![\text{Scalar}[(\text{Chiffre})^+(,(\text{Chiffre})^+)?]]\!] &= Tableau :\rightarrow [\![\text{Scalar}]\!] \end{aligned}$$

6.2 Variables

Les variables se déclarent en associant un identifiant (Id) à un type (Type) de la manière suivante :

VarDecl	::=	Id **as** Type

Les variables globales sont déclarées à la suite de la clause **declare and retain** et les déclarations sont terminées par des point-virgules (**;**). Nous verrons plus loin que les variables peuvent également être déclarées à un niveau local et que les déclaration globales peuvent inclure des fonctions. Limitons-nous pour l'instant aux variables et voyons comment donner leur sémantique. La sémantique d'une déclaration de variable revient à ajouter cette variable dans l'environnement courant et à réserver un espace pour cette variable dans la mémoire. Cela revient donc à mettre à jour l'état du programme $s = (\sigma, \epsilon, i, o)$ pour que ϵ retourne le type et l'adresse de cette nouvelle variable et que σ retourne la valeur stockée à cette adresse pour la variable. Afin de réserver l'espace mémoire adéquat, nous allons d'abord définir une fonction d'allocation *Alloc*, qui, pour un type donné et une mémoire donnée (σ), retourne une adresse ($\mathbb{A}$) et le contenu de la mémoire à cette adresse ($\mathbb{S}$) :

$$Alloc : \mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{A} \times \mathbb{S}.$$

Pour chaque type, la fonction est définie de la manière suivante, notez que les variables booléennes sont initialisées à $faux$ et les variables entières et de type case sont initialisées à 0 :

$$\begin{aligned} Alloc(\textbf{boolean})\sigma &= (a, \sigma[a/faux]) \text{ si } a \text{ est le plus petit naturel t.q. } \sigma(a) = \perp \\ Alloc(\textbf{integer})\sigma &= (a, \sigma[a/0]) \text{ si } a \text{ est le plus petit naturel t.q. } \sigma(a) = \perp \\ Alloc(\textbf{square})\sigma &= (a, \sigma[a/0]) \text{ si } a \text{ est le plus petit naturel t.q. } \sigma(a) = \perp \end{aligned}$$

Dans le cas d'un tableau, il faut trouver un ensemble d'adresses contigües qui ne sont pas encore utilisées dans l'environnement courant. Le nombre d'adresses est la taille du tableau. Tout comme les variables de type scalaire, les cases du tableau sont initialisées à une valeur par défaut ($faux$ pour les tableaux de booléens et 0 pour les tableaux d'entiers et de types de cases). À noter qu'un tableau de taille nulle est permis et déclenchera toujours une erreur lors de l'accès à l'une de ses cases.

Nous pouvons maintenant donner la sémantique d'une déclaration de variable comme étant une fonction qui, pour un état donné $s = (\sigma, \epsilon, i, o)$, met à jour cet état en réservant l'espace nécessaire et en initialisant la valeur de la variable :

$$[\![\text{Id } \textbf{as } \text{Type}]\!]s = (\sigma_1, \epsilon[\text{Id}/a], i, o), \text{ où } (a, \sigma_1) = Alloc(\text{Type})\sigma$$

6.3 Plateau de jeu

Le monde est représenté par un plateau de jeu carré de dimension fixe (**square** [(Chiffre)$^+$, (Chiffre)$^+$]), sur lequel les personnages se déplacent. Le plateau de joue est défini à partir de l'instruction suivante **arena as square** [(Chiffre)$^+$, (Chiffre)$^+$] où **arena** est le nom réservé utiliser pour déclarer le plateau et

square [(Chiffre)$^+$, (Chiffre)$^+$] est de type case. Concrètement, **arena** est une variable tableau de type tableau à deux dimensions de taille carrée. Le plateau de jeu (**arena**) est déclarée par défaut à deux dimensions de taille de 9 par 9 (**arena as square** [9, 9]). Le plateau de jeu doit être déclarée à la suite de la clause **declare and retain**.

6.4 Règles de nommage

Dans un programme *PILS*, les définitions suivantes sont interdites :

1. une variable globale ne peut porter le même nom qu'une autre variable globale ;
2. una variable globale différent de celui du plateau de joue ne peut porter le nom réservé **arena** ;
3. une variable locale ne peut porter le même nom qu'une autre variable locale ;
4. una variable locale différent de celui du plateau de joue ne peut porter le nom réservé **arena** ;
5. une variable locale ne peut porter le même nom qu'une autre variable locale ;
6. une variable locale ne peut porter le même nom qu'un des paramètres formels de la fonction dans laquelle elle est définie ;
7. une variable locale ne peut porter le même nom que la fonction dans laquelle elle est définie ;
8. une variable globale ne peut porter le même nom qu'une fonction ;
9. les paramètres formels d'une fonction ne peuvent porter le même que la fonction dans laquelle ils sont définis ;
10. deux paramètres formels d'une fonction ne peuvent porter le même nom ;
11. une fonction ne peut porter le même nom qu'une autre fonction.

Ces règles permettent à une variable locale et à un paramètre de porter le même nom qu'une variable globale. Dans ce cas, cet identificateur local masque la variable de niveau supérieur. De plus, l'identifiant d'une fonction est unique, la surcharge de fonction est interdite en *PILS* (*i.e.*, il ne peut exister deux fonctions avec le même identifiant, même si le nombre et/ou le type de leurs paramètres sont différents).

7 Expressions droites

Les expressions droites sont des expressions dont l'évaluation retourne une valeur (i.e. un élément de $\mathbb{V}$). Elles portent ce nom car elles apparaissent dans la partie *droite* des instructions d'affectation. Une expression droite peut modifier, par effet de bords, l'état courant. L'évaluation d'une expression peut échouer, parce que l'on se réfère à une variable entière qui n'existe pas, par exemple. La sémantique d'une expression droite sera donc donnée par une fonction *partielle*. Cette fonction sera donc définie si l'évaluation réussit. Il est particulièrement important de faire attention au *type* de ces expressions.

Une expression droite est soit une expression de type entière, case ou booléenne. On retrouve ci-dessous les expressions classiques : addition, soustraction, multiplication, division entière, opérations booléennes, appel de fonction, et accès à la valeur stockée à dans une variable. La grammaire suivante définit une expression droite (ExprD) comme étant une expression entière (ExprEnt), une expression booléenne (ExprBool), une expression portant sur les types de cases (ExprCase), une expression gauche (*i.e.*, désignant une variable ou une case de tableau), un appel de fonction ou une expression droite placée entre parenthèses :

```
ExprD   ::=  ExprEnt
           | ExprBool
           | ExprCase
           | ExprG
           | Id( (ExprD (,ExprD)*)? )
           | (ExprD)
```

Avant d'évaluer une expression, nous demandons que son type soit vérifié. Une erreur de type est une erreur de sémantique statique qui doit être détectée lors de l'analyse syntaxique. Un programme comprenant une erreur de type sera rejeté *avant exécution* par un compilateur *PILS*. Nous définissons les règles de typage générique à toute expression droite comme suit :

1. pour un appel de fonction Id(ExprD_1, ExprD_2, ... , ExprD_n), l'expression est bien typée s'il existe une fonction associée à l'Id donné dans l'environnement courant, le type de l'ExprD sera dans ce cas le type de retour de la fonction (**void** étant associé à un type indéfini *nil*) ;

2. pour une expression gauche ExprG, l'expression est bien typée si la variable désignée existe dans l'environnement courant ϵ. Dans le cas d'un tableau, la case désignée par l'expression gauche existe dans ϵ et est de type Scalar. Dans ce dernier cas, l'ExprG sera bien typée si la/les expression(s) droite(s) intervenant pour désigner la case du tableau (*e.g.*, Id[ExprD_1 , ExprD_2]) sont bien typées ; le type de l'ExprD sera dans ce cas le type de la variable désignée dans l'environnement courant ϵ ;
3. pour une expression droite entre parenthèses (ExprD), l'expression est bien typée si ExprD est bien typée ; le type de l'expression sera dans ce cas le type de l'ExprD placée entre parenthèses.

Enfin, la sémantique d'une expression droite est donnée par la fonction :

$$[\![.]\!] : \text{ExprD} \rightarrow \mathbb{S}state \nrightarrow \mathbb{R}$$

Pour rappel, $\mathbb{R}$ est défini comme $\mathbb{R} = \mathbb{V} \times \mathbb{S}state$, ce qui signifie donc que les résultats de l'application de la fonction sémantique seront des couples (v, s) où v représente une valeur et s un état du programme. On évalue les opérandes *de gauche à droite*, ce qui peut modifier (par effets de bord) l'état du programme. C'est pourquoi la deuxième opérande sera évaluée dans l'état résultant de l'évaluation de la première opérande. Enfin, la sémantique n'est définie que si la sémantique des composants est définie. Ce qui signifie que en cas de "plantage", l'évaluation d'une expression droite est donc indéfinie (d'où la définition d'une fonction partielle). Nous donnons les sémantiques des éléments communs à toutes les expressions droites pour un programme dans un état $s = (\sigma, \epsilon, i, o)$ comme suit :

Expressions gauches La sémantique d'une expression gauche ExprG est définie en accédant à la case mémoire correspondante (noté $\sigma(a)$) et dont l'adresse (a) est donnée par la fonction sémantique $[\![.]\!]_G$ appliquée à cette expression gauche et retournant une adresse (*cf.* Section 8) :

$$[\![\text{ExprG}]\!]s = (\sigma(a), s_1), \text{ où } [\![\text{ExprG}]\!]_G s = (a, s_1)$$

Si l'expression gauche désigne une case de tableau, la fonction sémantique $[\![\text{Id[ExprD(,ExprD)?]}]\!]_G$ commencera par évaluer la/les expression(s) droite(s) ExprD avant de déterminer l'adresse de la case du tableau. L'évaluation des ExprD se faisant de gauche à droite.

Appel de fonction La sémantique d'un appel de fonction $\text{Id(ExprD}_1, \ldots, \text{ExprD}_n\text{)}$ revient à aller chercher la sémantique de cette fonction f dans l'environnement courant $f = \epsilon(\text{Id}, fct)$, à mettre à jour l'environnement courant afin d'y ajouter le contexte de la fonction, et, pour chaque paramètre, associer la valeur résultant de l'évaluation de ExprD_i au nom de ce paramètre :

$$[\![\text{Id(ExprD}_1, \ldots, \text{ExprD}_n\text{)}]\!]s = f\ s_n\ (v_1, \ldots, v_n)$$

Où f est la fonction sémantique associée à Id prenant en entrée un état s_n et un ensemble de valeurs pour les différents paramètres, s_n est l'état résultant de l'évaluation des expressions droites ExprD_i (elles aussi évaluées de gauche à droite) donnant les valeurs des différents paramètres v_i. En d'autres termes, $[\![ExprD_1]\!]s = (v_1, s_1)$, $[\![ExprD_2]\!]s_1 = (v_2, s_2)$, ..., $[\![ExprD_n]\!]s_{n-1} = (v_n, s_n)$.

Expression parenthésée Enfin, la sémantique d'une expression entière placée entre parenthèses revient à la sémantique de cette expression entière :

$$[\![\text{(ExprD)}]\!]s = [\![\text{ExprD}]\!]s$$

7.1 Expressions entières

Une expression entière (ExprEnt) est définie comme étant un entier, une variable de l'environnement (**latitude**, **grid size**, *etc.*) ou toute opération portant sur deux expressions elles-mêmes entières :

```
ExprEnt  ::=  Entier
           | latitude | longitude | grid size
           | (map | radio | ammo | fruits | soda) count
           | life
           | ExprD+ ExprD
           | ExprD- ExprD
           | ExprD* ExprD
           | ExprD/ ExprD
           | ExprD% ExprD
```

7.1.1 Vérification des types

Une expression entière (ExprEnt) sera bien typée si :

1. il s'agit d'une constante Entier ou d'une variable d'environnement **latitude**, **longitude**, **grid size**, (**map** | **radio** | **ammo** | **fruits** | **soda**) **count**, **life** ;
2. pour une opération sur deux expressions entières ExprD_1 (+ | - | * | / | %) ExprD_2, l'expression est bien typée si les expressions ExprD_1 et ExprD_2 sont bien typées et de type entier.

7.1.2 Sémantique

Nous pouvons maintenant donner la sémantique des différents types d'expressions entières pour un programme dans un état $s = (\sigma, \epsilon, i, o)$.

Entiers La sémantique d'un Entier est directe et donnée par :

$$[\![Entier]\!]s = ([\![Entier]\!], s)$$

Variables d'environnement Une variable d'environnement **latitude**, **longitude**, **grid size**, (**map** | **radio** | **ammo** | **fruits** | **soda**) **count** ou **life** est définie (comme leur nom l'indique) dans ϵ et est accessible à chaque endroit du programme. Sa valeur est gérée au niveau global et évolue à chaque tour de jeu. La sémantique d'une variables d'environnement v revient donc (comme pour les autres variables) à aller chercher la valeur stockée en mémoire à l'adresse associée à la variable, retrouvée grâce à l'environnement ϵ (que l'on notera $\epsilon(v, vp)$, où vp est la catégorie *variable de programme*) :

$$[\![v]\!]s = (\sigma(\epsilon(v, vp)), s)$$

La signification des différentes variables est donnée ci-dessous :

- **longitude** donne la position absolue X de l'EA par rapport à l'origine située en bas à gauche de la grille de jeu ;
- **latitude** donne la position absolue Y de l'EA par rapport à l'origine située en bas à gauche de la grille de jeu ;
- **grid size** donne la longueur de la grille de jeu (qui est carrée) en nombre de cases, la **longitude** et la **latitude** sont donc compris entre 0 et **grid size** $- 1$;
- (**map** | **radio** | **ammo** | **fruits** | **soda**) **count** donnent respectivement le nombre de cartes, radios, munitions, fruits et boissons à haute teneur en sucre que l'EA possède dans son inventaire (l'on retrouve cette information dans le bas de l'écran de jeu, cf. Figure 3) ;
- **life** indique le nombre de points de vie restants de l'EA, **life** étant compris entre 0 et 100 (l'EA étant irrémédiablement perdue si le nombre de points de vie tombe à 0, la partie sera perdue), cette information se retrouve également dans le bas de l'écran de jeu (cf. Figure 3).

Opérateurs binaires Pour un opérateur binaire +, -, *, / ou %, la sémantique revient à évaluer l'opérande gauche, puis l'opérande droite, puis à effectuer l'opération mathématique sur les résultats :

$$\begin{aligned}
[\![\mathsf{ExprD}_1 + \mathsf{ExprD}_2]\!]s &= (v_1 + v_2, s_2), \text{ où } [\![\mathsf{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\mathsf{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\mathsf{ExprD}_1 - \mathsf{ExprD}_2]\!]s &= (v_1 - v_2, s_2), \text{ où } [\![\mathsf{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\mathsf{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\mathsf{ExprD}_1 * \mathsf{ExprD}_2]\!]s &= (v_1 * v_2, s_2), \text{ où } [\![\mathsf{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\mathsf{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\mathsf{ExprD}_1 / \mathsf{ExprD}_2]\!]s &= (v_1/v_2, s_2), \text{ où } [\![\mathsf{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\mathsf{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\mathsf{ExprD}_1 \% \mathsf{ExprD}_2]\!]s &= (v_1 \% v_2, s_2), \text{ où } [\![\mathsf{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\mathsf{ExprD}_2]\!]s_1 = (v_2, s_2)
\end{aligned}$$

7.2 Expressions booléennes

Une expression booléenne (ExprBool) est définie de la manière suivante :

```
ExprBool  ::=  true | false
             | ennemi is ( north | south | east | west)
             | graal is ( north | south | east | west)
             | ExprD and ExprD
             | ExprD or ExprD
             | not ExprD
             | ExprD < ExprD
             | ExprD > ExprD
             | ExprD = ExprD
```

7.2.1 Vérification des types

Une expression booléenne (ExprBool) sera bien typée si :

1. il s'agit d'une constante **true**, **false**, **ennemi is** (**north** | **south** | **east** | **west**), **graal is** (**north** | **south** | **east** | **west**) ;
2. pour une opération sur deux expressions booléennes ExprD_1 (**and** | **or**) ExprD_2, l'expression est bien typée si les expressions ExprD_1 et ExprD_2 sont bien typées et de type booléen ;
3. pour la négation d'une expression booléenne **not** ExprD, l'expression est bien typée si ExprD est bien typée et de type booléen ;
4. pour les opérateurs de comparaison travaillant sur deux expressions ExprD_1 ($<$ | $>$) ExprD_2, l'expression est bien typée si les expressions ExprD_1 et ExprD_2 sont bien typées et de type entier ;
5. pour l'égalité entre deux expressions $\text{ExprD}_1 = \text{ExprD}_2$, l'expression est bien typée si les expressions ExprD_1 et ExprD_2 sont bien typées et sont *toutes les deux* de type entier, booléen ou type de case ;

7.2.2 Sémantique

Tout comme pour les expressions entières, nous pouvons donner la sémantique des différents types d'expressions booléennes pour un programme dans un état $s = (\sigma, \epsilon, i, o)$.

Constantes La sémantique d'une constante **true** ou **false** est directe et donnée par :

$$[\![\textbf{true}]\!]s = (vrai, s)$$
$$[\![\textbf{false}]\!]s = (faux, s)$$

Variables d'environnement Une variable d'environnement **ennemi is** (**north** | **south** | **east** | **west**), **graal is** (**north** | **south** | **east** | **west**) est définie dans ϵ et est accessible à chaque endroit du programme. Sa valeur est gérée au niveau global et évolue à chaque tour de jeu. Tout comme pour les variables d'environnement entières, la sémantique d'une variable d'environnement v est donnée par :

$$[\![v]\!]s = (\sigma(\epsilon(v, vp)), s)$$

La valeur de ces variables est modifiée lorsque l'EA utilisera une **radio** ou une **map** et indiqueront où trouver l'**ennemi** (*i.e.*, l'EA adverse) ou le **graal** au moment de cette utilisation (attention, la valeur n'est donc pas mise à jour en fonction des mouvements de l'adversaire). Ces valeurs sont donc toutes à $faux$ tant que l'EA n'a pas utilisé de **radio** ou de **map**.

Opérateurs binaires Pour un opérateur binaire **and** ou **or**, la sémantique revient à évaluer l'opérande gauche, puis l'opérande droite, puis à effectuer l'opération mathématique sur les résultats :

$$[\![\text{ExprD}_1 \textbf{ and } \text{ExprD}_2]\!]s = (v_1 \wedge v_2, s_2), \text{ où } [\![\text{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\text{ExprD}_2]\!]s_1 = (v_2, s_2)$$
$$[\![\text{ExprD}_1 \textbf{ or } \text{ExprD}_2]\!]s = (v_1 \vee v_2, s_2), \text{ où } [\![\text{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\text{ExprD}_2]\!]s_1 = (v_2, s_2)$$

Opérateur unaire Pour un opérateur unaire **not**, la sémantique revient à évaluer l'opérande puis à effectuer l'opération mathématique sur les résultats :

$$[\![\textbf{not } \text{ExprD}]\!]s = (\neg v, s_1), \text{ où } [\![\text{ExprD}]\!]s = (v, s_1)$$

FIGURE 3 – Partie de la grille visible pour un EA et repérage sur celle-ci.

Opérateurs de comparaison Pour un opérateur de comparaison binaire $<$, $>$ ou $=$, la sémantique revient à évaluer l'opérande gauche, puis l'opérande droite, puis à effectuer l'opération mathématique sur les résultats :

$$\begin{aligned}
[\![\text{ExprD}_1 < \text{ExprD}_2]\!]s &= (v_1 < v_2, s_2), \text{ où } [\![\text{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\text{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\text{ExprD}_1 > \text{ExprD}_2]\!]s &= (v_1 > v_2, s_2), \text{ où } [\![\text{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\text{ExprD}_2]\!]s_1 = (v_2, s_2) \\
[\![\text{ExprD}_1 = \text{ExprD}_2]\!]s &= (v_1 = v_2, s_2), \text{ où } [\![\text{ExprD}_1]\!]s = (v_1, s_1) \text{ et } [\![\text{ExprD}_2]\!]s_1 = (v_2, s_2)
\end{aligned}$$

7.3 Expressions sur les types de cases

Une expression sur les types de cases (ExprCase) est définie de la manière suivante :

ExprCase	::=	**dirt** | **rock** | **vines** | **zombie** | **player** | **ennemi** | **map** | **radio** | **ammo** | **fruits** | **soda** | **graal** | **nearby**[ExprD,ExprD]

L'expression **nearby**[ExprD,ExprD] permet de consulter le contenu des cases adjacentes au personnage. Concrètement, **nearby** est une variable d'environnement de type tableau de types de cases à deux dimensions de taille carrée de 9 par 9. Lors de l'exécution d'un programme $PILS$, le personnage se trouvera toujours en position $[4, 4]$, le premier indice donnant la coordonnée X et le second indice donnant la coordonnée Y. Le repérage sur une grille se fera comme décrit dans la Figure 3.

7.3.1 Vérification des types

Une expression de type case (ExprCase) sera bien typée si : il s'agit d'une constante **dirt**, **rock**, **vines**, **zombie**, **player**, **ennemi**, **map**, **radio**, **ammo**, **fruits**, **graal** ou **soda** ; dans le cas de la consultation des cases adjacentes, les indices donnés dans l'expression **nearby**[ExprD,ExprD] sont de type entier (comme pour les tableaux, un indice dépassant les bornes provoquera une erreur lors de l'*exécution* du programme, pas lors de sa compilation).

7.3.2 Sémantique

À nouveau, nous pouvons donner la sémantique des différents types d'expressions case pour un programme dans un état $s = (\sigma, \epsilon, i, o)$. La sémantique des constantes, comme pour les expressions booléennes et entières est directe. Les valeurs sémantiques associées aux types de cases sont définie dans l'ensemble $\mathbb{C}$, contenant les entiers compris entre 0 et 10. La fonction sémantique retournera donc une valeur entière pour chaque constante :

- $[\![\textbf{dirt}]\!]s = (0, s)$, de la poussière sur laquelle l'EA peut passer ;
- $[\![\textbf{rock}]\!]s = (1, s)$, un rocher bloquant le passage (la case ne peut donc pas être accédée) ;
- $[\![\textbf{vines}]\!]s = (2, s)$, des ronces que l'EA peut couper à l'aide de sa machette en avançant vers cette case plusieurs fois (en fonction de la densité du buisson de ronces) ;
- $[\![\textbf{zombie}]\!]s = (3, s)$, un zombie qui attaquera l'EA à portée (et qui le suivra pour lui faire tout un tas de trucs pas recommandés pour la santé si l'EA passe trop près) ;
- $[\![\textbf{ennemi}]\!]s = (4, s)$, l'EA adverse ;
- $[\![\textbf{player}]\!]s = (5, s)$, l'EA (toujours située au centre de l'écran de jeu) ;
- $[\![\textbf{map}]\!]s = (6, s)$, une carte (objet **map**) que l'EA peut ramasser en passant sur la case ;
- $[\![\textbf{radio}]\!]s = (7, s)$, une radio (objet **radio**) que l'EA peut ramasser en passant sur la case ;
- $[\![\textbf{ammo}]\!]s = (8, s)$, des munitions (objet **ammo**) que l'EA peut ramasser en passant sur la case ;
- $[\![\textbf{fruits}]\!]s = (9, s)$, des fruits (objet **fruits**) que l'EA peut ramasser en passant sur la case ;
- $[\![\textbf{soda}]\!]s = (10, s)$, une boisson à haute teneur en sucre (objet **soda**) que l'EA peut ramasser en passant sur la case ;
- $[\![\textbf{graal}]\!]s = (11, s)$, le graal c'est un objet magique qui ouvre l'accès au monde suivant.

Pour les cases entourant le personnage, la sémantique $[\![\textbf{nearby}[long, lat]]\!]s$ sera la même que pour la sémantique d'une expression droite désignant une case de tableau.

7.4 Priorité et associativité des opérateurs

Afin de mettre un programme *PILS* sous une forme non-ambigüe, il faut connaître les règles de précédence et d'associativité des différents opérateurs. Ces règles permettent de définir dans quel ordre seront évaluées les différentes opérations. Nous choisissons la règle de précédence suivante :

$$\{\textbf{not}\} < \{\textbf{and}, \textbf{or}\} < \{=, <, >\} < \{+, -\} < \{*, /, \%\} < \{(.)\}$$

Où $b < a$ signifie a doit être évalué avant b. Toutes les opérations sont associatives à gauche : on lit les termes de gauche à droite. Par exemple, l'expression : $\mathbf{x + 3 * y - 5 - z}$ doit être interprétée comme $\Big(\big(\mathbf{x}+(\mathbf{3}*\mathbf{y})\big)-\mathbf{5}\Big)-\mathbf{z}$.

Un autre exemple : **true and a and true** sera interprété comme : (**true and a**) **and true**. Un troisième exemple plus complexe :

not x = 81∗712 and true or y>x

Sera interprété comme :

$$\textbf{not}\Big(\Big(\big(\mathbf{x}=(\mathbf{81}*\mathbf{712})\big)\textbf{and true}\Big)\ \textbf{or}\ (\mathbf{y}>\mathbf{x})\Big)$$

8 Expressions gauches

Une expression gauche sert à référencer une variable ou une case de tableau, ceux-ci pouvant être à plusieurs dimensions. Elle est définie par la grammaire :

```
ExprG   ::=   Id
            | Id[ExprD (, ExprD)? ]
```

Une expression gauche sera correctement typée si, au moment de son évaluation, il existe une variable dans le contexte courant ϵ portant l'Id renseigné. Et, dans le cas d'un tableau, si l'ExprD est correctement typée et de type entière. Intuitivement, l'identifiant *doit* avoir été déclaré dans le contexte où il est utilisé. En *PILS*, cela n'est possible que si l'identifiant est :

1. un paramètre de la fonction (cf. Section 10) ;

2. le nom de la fonction courante, auquel cas il s'agira de la valeur de retour de la fonction (cf. Section 10) ;
3. une variable locale au bloc d'exécution en cours (locale à la fonction ou à la ClauseWhen dans lequel la variable est référencée, cf. Sections 10 et 12) ;
4. une variable globale déclarée après les mots clés **declare and retain** (cf. Section 12).

Les règles de visibilité d'une variable pour un Id donné correspondront à l'ordre donné ci-dessus : une variable a booléenne déclarée localement à une fonction cachera donc une éventuelle variable a (potentiellement d'un autre type) déclarée globalement après les mots clés **declare and retain**).

La sémantique d'une expression gauche est donnée par la fonction $[\![\mathsf{ExprG}]\!]_G \rightarrow \mathbb{S}tate \hookrightarrow \mathbb{A} \times \mathbb{S}tate$, retournant pour une expression gauche donnée son adresse et l'éventuel nouvel état résultant de l'évaluation des expressions entières dans le cas d'un tableau. Pour un programme dans un état $s = (\sigma, \epsilon, i, o)$, l'on a donc :

$$\begin{aligned} [\![\mathsf{Id}]\!]_G s &= (\epsilon(\mathsf{Id}), s) \\ [\![\mathsf{Id}[\mathsf{ExprEnt}_1(,\mathsf{ExprEnt}_2)?]]\!]_G s &= [\![\mathsf{Id}]\!]_G s_2 + \textit{offset}, \text{ où } \textit{offset} \text{ est calculé sur base de} \\ &\quad [\![\mathsf{ExprEnt}_1]\!], [\![\mathsf{ExprEnt}_2]\!] \end{aligned}$$

L'on remarque donc que l'adresse d'une case d'un tableau correspond à l'adresse de base du tableau à laquelle s'ajoute un *offset*, dépendant des résultats des évaluations des expressions entières $\mathsf{ExprEnt}_i$. En cas de tableau à 2 dimensions, l'on commencera par évaluer l'expression entière la plus à gauche ($\mathsf{ExprEnt}_1$) et l'on terminera par l'expression entière la plus à droit ($\mathsf{ExprEnt}_2$) avant de calculer l'offset.

9 Instructions

Les instructions reprennent les instructions classiques d'un langage de programmation impératif : l'instruction vide (**skip**), l'instruction conditionnelle, la boucle, l'affectation, le calcul d'une expression droite (ce qui permet notamment les appels de fonctions void) et enfin, l'instruction **next** qui interrompra l'exécution et indiquera la prochaine action à effectuer (pratiquement, l'instruction **next** a le même effet qu'un `exit(`n`)` placé dans le programme principal que l'on retrouve dans d'autres langages de programmation). Les instructions sont définies par la grammaire suivante :

```
Instruction ::= skip
              | if ExprD then Instruction + done
              | if ExprD then Instruction + else Instruction + done
              | while ExprD do Instruction + done
              | set ExprG to ExprD
              | compute ExprD
              | next Action
```

Les actions possibles comprennent le déplacement, l'utilisation d'une arme à projectiles, l'utilisation d'une *item* ou le passage d'un tour de jeu :

```
Action ::= move ( north | south | east | west)
         | shoot ( north | south | east | west)
         | use ( map | radio | fruits | soda)
         | do nothing
```

9.1 Vérification des types

Tout comme pour les expressions droites, nous imposons les vérifications de types suivantes pour les instructions :

Instruction vide L'instruction **skip** sera toujours bien typée.

Conditionnelles Les instructions **if** ExprD **then** Instruction $^+$ **done** et **if** ExprD **then** Instruction $^+$ **else** Instruction $^+$ **done** sont bien typées si l'ExprD est bien typée et de type booléen et si la/les branche(s) Instruction $^+$ sont bien typées.

Boucle L'instruction **while** ExprD **do** Instruction $^+$ **done** est bien typée su l'ExprD est bien typée et de type booléen et si la/les dans Instruction $^+$ sont bien typées.

Affectation L'instruction **set** ExprG **to** ExprD est bien typée si ExprG et ExprD sont bien typées, si le type de ExprG est le même que celui de ExprD et si ExprG et ExprD sont de type scalaire (Scalar).

Calcul d'une expression droite L'instruction **compute** ExprD sera bien typée si ExprD est bien typée.

Indication de la prochaine action de jeu L'instruction **next** Action sera toujours bien typée par construction.

9.2 Sémantique

La sémantique des instructions est donnée par la fonction :

$$[\![.]\!] : \mathsf{Instruction} \to \mathbb{S}tate \nrightarrow (\mathbb{R}es \cup \mathbb{S}tate \cup \mathbb{O})$$

Une instruction peut donc soit délivrer un résultat (*i.e.*, une valeur et un nouvel état), soit délivrer un état intermédiaire, soit retourner une sortie du programme et arrêter son exécution (dans le cas de l'instruction **next** Action). Pour un programme dans un état $s = (\sigma, \epsilon, i, o)$, la sémantique des différentes instructions est :

Instruction vide La sémantique de l'instruction vide ne modifie par l'état courant :

$$[\![\textbf{skip}]\!]s = s$$

Conditionnelles Les instructions conditionnelles commenceront par évaluer la condition et exécuteront la branche **then** ou **else** (si celle-ci est définie). Nous définirons la sémantique de

$$[\![\textbf{if } \mathsf{ExprD} \textbf{ then } \mathsf{Instruction}^+ \textbf{done}]\!]$$

comme étant

$$[\![\textbf{if } \mathsf{ExprD} \textbf{ then } \mathsf{Instruction}^+ \textbf{else skip done}]\!]$$

La sémantique de l'instruction conditionnelle est donc :

$$[\![\textbf{if } \mathsf{ExprD} \textbf{ then } \mathsf{Instruction}_1^+ \textbf{else } \mathsf{Instruction}_2^+ \textbf{ done}]\!] = \begin{cases} [\![\mathsf{Instruction}_1^+]\!]s_1 & \text{si } [\![\mathsf{ExprD}]\!]s = (vrai, s_1) \\ [\![\mathsf{Instruction}_2^+]\!]s_1 & \text{si } [\![\mathsf{ExprD}]\!]s = (faux, s_1) \end{cases}$$

Boucle La sémantique de la boucle est définie de manière récursive, l'on commence par évaluer la condition, si celle-ci est fausse, la boucle s'arrête (dans l'état s_1), sinon l'on évalue le corps de la boucle (sur l'état l'état s_1) avant de passer à l'itération suivante :

$$[\![\textbf{while } \mathsf{ExprD} \textbf{ do } \mathsf{Instruction}^+ \textbf{done}]\!]s = \begin{cases} s_1 \text{ si } [\![ExprD]\!]s = (faux, s_1) \\ [\![\textbf{while } \mathsf{ExprD} \textbf{ do } \mathsf{Instruction}^+ \textbf{done}]\!][\![\mathsf{Instruction}^+]\!]s_1 \\ \quad \text{si } [\![ExprD]\!]s = (vrai, s_1) \end{cases}$$

Affectation L'affectation d'une expression droite à une expression gauche procède en deux étapes. D'abord, l'expression gauche et l'expression droite sont évaluées dans l'état courant. Ensuite, si ces évaluations donnent un résultat, on remplace la valeur stockée en mémoire à l'adresse calculée, par la valeur de l'expression droite. On commence par évaluer l'expression gauche, ensuite on évalue l'expression droite :

$$[\![\textbf{set } \mathsf{ExprG} \textbf{ to } \mathsf{ExprD}]\!] = (\sigma_2[a/v], \epsilon_2, i_2, o_2) \quad \text{où} \quad \begin{array}{l} [\![\mathsf{ExprG}]\!]_G s = (a, s_1) \\ [\![\mathsf{ExprD}]\!]s_1 = (v, s_2) \end{array}$$

Calcul d'une expression droite La sémantique associée à l'instruction **compute** revient à calculer la valeur de l'expression droite et à jeter cette valeur avant de continuer l'exécution. Si l'expression est un appel à une fonction retournant une valeur de type void, l'on considèrera que la valeur de retour n'est pas définie. La sémantique est donc simplement :

$$[\![\textbf{compute}\ \mathsf{ExprD}]\!]s = s_1, \text{ où } [\![\mathsf{ExprD}]\!]s = (v, s_1)$$

Indication de la prochaine action de jeu L'indication de la prochaine action à effectuer par le personnage a 2 effets : écrire sur le flux de sortie l'entier correspondant à l'action que l'on désire effectuer (et dont la correspondance entre une action et un entier est donnée ci-dessous) et arrêter l'exécution du programme.

- $[\![$**next do nothing**$]\!]s = 0$, l'EA ne fera rien ;
- $[\![$**next move north**$]\!]s = 1$, l'EA se déplacera d'une case vers le nord si c'est possible ;
- $[\![$**next move east**$]\!]s = 2$, l'EA se déplacera d'une case vers l'est si c'est possible ;
- $[\![$**next move south**$]\!]s = 3$, l'EA se déplacera d'une case vers le sud si c'est possible ;
- $[\![$**next move west**$]\!]s = 4$, l'EA se déplacera d'une case vers l'ouest si c'est possible ;
- $[\![$**next shoot north**$]\!]s = 5$, l'EA tirera vers le nord s'il a encore des munitions, sinon il ne fait rien ;
- $[\![$**next shoot east**$]\!]s = 6$, l'EA tirera vers l'est s'il a encore des munitions, sinon il ne fait rien ;
- $[\![$**next shoot south**$]\!]s = 7$, l'EA tirera vers le sud s'il a encore des munitions, sinon il ne fait rien ;
- $[\![$**next shoot west**$]\!]s = 8$, l'EA tirera vers l'ouest s'il a encore des munitions, sinon il ne fait rien ;
- $[\![$**next use map**$]\!]s = 9$, l'EA utilisera un objet carte qui lui indiquera dans quelle direction trouver le GRAAL (les constantes **graal is** (**north** | **south** | **east** | **west**) seront mises à jour au prochain tour) ;
- $[\![$**next use radio**$]\!]s = 10$, l'EA utilisera un objet radio qui lui indiquera dans quelle direction trouver l'EA ennemi (les constantes **ennemi is** (**north** | **south** | **east** | **west**) seront mises à jour au prochain tour) ;
- $[\![$**next use fruits**$]\!]s = 11$, l'EA mangera des baies qui lui redonneront beaucoup de santé (**life**) ;
- $[\![$**next use soda**$]\!]s = 12$, l'EA boira une boisson à haute teneur en sucre qui lui redonnera un peu de santé (**life**).

10 Déclarations de fonctions

Les fonctions sont déclarées au niveau global d'un programme *PILS* (uniquement). La syntaxe est définie de la manière suivante :

FctDecl	::=	Id **as function (** (VarDecl (**,**VarDecl)*)? **):** (Scalar | **void**) (**declare local** (VarDecl**;**)$^+$) ? **do** (Instruction)$^+$ **return** (ExprD | **void**) **done**

Les règles de nommage décrites à la Section 6.4 sont d'application ici pour l'identifiant de la fonction, les identifiants de paramètres et de variables locales. Une fois déclarée, une fonction peut être invoquée depuis une autre fonction (quelque soit l'ordre de déclaration de ces fonctions) ou depuis une clause **when**.

10.1 Vérification des types

Afin de faciliter l'écriture des règles de sémantique portant sur les procédures et d'en augmenter la lisibilité, nous utiliserons dans la suite de cette section une notation "compacte" de la déclaration de fonction. Une fonction *PILS* f avec des paramètres $p_1, \ldots, p_n$ de types $t_1, \ldots, t_n$ et retour une valeur de type t sera notée :

$$f\ (t_1 p_1, \ldots, t_n p_n)\ dvl\ il\ return\ t\ done$$

Avec :

- *dvl* les déclarations de variables,
- *il* la liste des instructions de la fonction,
- *return* le retour une valeur de type,
- *done* la fin d'exécution de la fonction et le retour à l'appelant.

La fonction f est bien typée si :

1. le type de retour t est soit **void**, soit entier, booléen ou de type case (*i.e.*, Scalar) ;
2. les types des paramètres sont entier, booléen ou de type case ;
3. les déclarations de variables de *dvl* sont correctes (respectent les contraintes décrites à la Section 6) ;
4. les instructions de *il* sont bien typées.

Le type de la fonction tel que renseigné dans l'environnement ϵ et utilisé lors des appels est t. Le type **void** est considéré comme le type non défini *nil*, une fonction de type void ne pourra donc être appelée que si sa valeur de retour n'est pas utilisée (dans une instruction **compute** ExprD).

10.2 Sémantique

La déclaration d'une fonction doit ajouter, dans l'état courant, la fonction sémantique qui y est associé. La sémantique d'une déclaration de fonction $PILS$ est calculée par la fonction :

$$[\![.]\!] : \mathsf{FctDecl} \rightarrow (\mathbb{S}tate \rightarrow \mathbb{S}tate)$$

Et est définie de la manière suivante pour une état $s = (\sigma, \epsilon, i, o)$:

$$[\![f\ (t_1p_1, \ldots, t_np_n)\ dvl\ il\ return\ t\ done]\!]s = (\sigma, \epsilon[f/[\![t(t_1p_1, \ldots, t_np_n)dvl\ il\ return\ done]\!]], i, o)$$

Où $[\![t(t_1p_1, \ldots, t_np_n)dvl\ il\ return\ done]\!]$ est intuitivement ce que le programme va faire lors de l'appel à la fonction (la sémantique de la fonction), qui retournera donc un élément de $\mathbb{F}$ (l'ensemble des sémantiques de fonctions). L'exécution d'une fonction peut être décomposée en plusieurs étapes :

1. Création d'un *environnement local*, au-dessus de la pile des environnements. Cet environnement comprendra par défaut une variable spéciale correspondant au nom de la fonction. La valeur affectée à cette variable lors de l'exécution de la fonction sera retournée à l'appelant après exécution de la fonction. Par exemple, l'on aura :

```
declare and retain
import inputFile.wld

        t as boolean;
        f as function(): boolean

        do

                set t to true
                /* other instructions here */
        return t;
        done
when your turn
        by default /* Default */
        declare local
                b as boolean;
        do
                /* b is false by default */
                set b to f()
                /* b is now true */
                next move west
        done
```

   La fonction sémantique $[\![t(t_1p_1, \ldots, t_np_n)dvl\ il\ done]\!]$ est donc définie de la manière suivante :

$$[\![t(t_1p_1, \ldots, t_np_n)dvl\ il\ done]\!]s = [\![(t_1p_1, \ldots, t_np_n)dvl\ il\ done]\!](\sigma_1, \epsilon \cdot c, i, o)$$

   Où $c = [f_{ret} \mapsto a]$ désigne le nouvel environnement, $(a, \sigma_1) = Alloc(t)\sigma$ et f_{ret} désigne la valeur de retour de la fonction f (afin de faire la distinction avec f également déclarée dans ϵ). L'état lors de l'exécution de la fonction devient dont $s = (\sigma, \epsilon \cdot c, i, o)$.

2. Initialisation des cellules mémoires correspondant aux paramètres formels avec les valeurs booléennes, entières ou de type de case. Les valeurs de ces paramètres sont évaluées dans l'environnement de l'appelant *avant* exécution de la fonction. Lors de l'exécution de la fonction, l'on disposera donc d'une liste de valeurs (parfois appelée paramètres effectifs pe) qui seront utilisées pour initialiser les différents paramètres.

$$[\![(t_1p_1, \ldots, t_np_n)dvl\ il\ done]\!]s(pe_1, \ldots, pe_n) = [\![(t_2p_2, \ldots, t_np_n)dvl\ il\ done]\!]s_1(pe_2, \ldots, pe_n)$$

Où $s_1 = (\sigma_1[a/pe_1], \epsilon \cdot c[p_1/a], i, o)$ et $(a, \sigma_1) = Alloc(t)\sigma$. À la différence d'une allocation classique, la valeur est ici initialisée en utilisant la valeur du paramètre effectif. Cette définition sémantique est inductive jusqu'à ce que tous les paramètres aient été créés dans l'environnement de la fonction.

3. Définition des variables locales (et leur initialisation à la valeur par défaut). La sémantique des déclarations de variables est elle aussi inductive jusqu'à ce que toutes les variables aient été déclarées (cf. Section 6) :

$$[\![dvl\ il\ done]\!]s = [\![il\ done]\!][\![dvl]\!]s$$

4. Exécution des instructions de la fonction de manière classique :

$$[\![il\ done]\!]s = [\![done]\!][\![il]\!]s$$

5. La fin de fonction déclenche la suppression de l'environnement local et le nettoyage de la mémoire avec retour de la valeur de la fonction à l'appelant. La sémantique $[\![done]\!]$ retournera donc un résultat $\in \mathbb{R}$:

$$[\![done]\!](\sigma, \epsilon \cdot c, i, o) = (\sigma(a), (\sigma, \epsilon, i, o)), \text{ où } (a, s) = [\![f_{ret}]\!]_G s$$

Une fonction déclarée comme **void** retournera une valeur indéfinie *nil* dans le contexte de l'appelant (la vérification des types garantissant que cette valeur n'est jamais utilisée par l'appelant, cela ne déclenchera pas d'erreur lors de l'exécution du programme). Une fonction non **void** retournera un résultat qui aura été stocké dans l'environnement et dont la valeur aura été définie au moyen d'une instruction d'affectation classique [3]. **Attention** : *la (non-)présence d'une valeur effective de retour ne doit pas être vérifiée par le compilateur*. Autrement dit, si pour une fonction non **void**, sa valeur de retour n'a pas été initialisée dans le corps de la fonction, cela déclenchera une erreur lors de l'*exécution* du programme (pour autant que cette valeur de retour ait été utilisée dans une ExprD), pas au moment de sa compilation. La détection de la présence d'une valeur de retour ou non dans une fonction au moment de la compilation est un problème non décidable.

6. Enfin, un système de *garbage collection* s'assurera de récupérer toutes les cellules mémoires inoccupées dans σ.

11 Déclaration de import

L'importation d'un fichier d'initialisation des valeurs des différentes variables d'environnement est déclarées au niveau global d'un programme $PILS$ (uniquement). La syntaxe est définie de la manière suivante :

ImpDecl	::=	**import** FileDecl

Les FileDecl est déclarent de la manière suivante :

FileDecl	::=	FileName.wld
FileName	::=	Lettre (Chiffre | Lettre)*

Par exemple :

```
declare and retain
  import inputFile.wld
  t as boolean;
  f as function(): boolean
        do
                set t to true
```

3. Ce qui devrait rappeler des souvenirs à ceux qui ont eu la chance de programmer en Pascal.

```
                /* other instructions here */
        return t;
        done
when your turn
        by default /* Default */
        declare local
                b as boolean;
        do
                /* b is false by default */
                set b to f()
                /* b is now true */
                next move west
        done
```

12 Programmes

Finalement, en combinant les éléments et grammaires précédents, nous pouvons défini deux types des programmes $PILS$:

1. **Décrivant les mondes** : Un programme $PILS$ est défini de la manière suivante comme un bloc de déclarations globales (variables et/ou fonctions) et comme une suite des instructions et clause par défaut. La grammaire est définie de la manière suivante :

   ```
   Programme   ::=  declare and retain
                      (VarDecl ; | FctDecl)*
                      (Instruction)*
                      ClauseDefault
   ```

   La clause par défaut sera toujours présente dans un programme $PILS$ et suivra la même structure que les clauses **when** :

   ```
   ClauseDefault   ::=  by default
                          (declare local (VarDecl ;)+) ?
                          do (Instruction)+ done
   ```

2. **Décrivant la stratégie** : Un programme $PILS$ est défini de la manière suivante comme un bloc de déclarations globales (variables et/ou fonctions) et comme une suite de clauses when terminée par une clause par défaut. La grammaire est définie de la manière suivante :

   ```
   Programme   ::=  declare and retain
                      (VarDecl ; | FctDecl | ImpDecl)*
                    when your turn
                      (ClauseWhen)*
                      ClauseDefault
   ```

   Une clause **when** se compose d'une expression booléenne spécifiant quand la clause sera exécutée, d'une éventuelle liste de déclarations de variables locales et d'un bloc d'instructions :

   ```
   ClauseWhen   ::=  when ExprD
                       (declare local (VarDecl ;)+) ?
                       do (Instruction)+ done
   ```

   La clause par défaut sera toujours présente dans un programme $PILS$ et suivra la même structure que les clauses **when** :

   ```
   ClauseDefault   ::=  by default
                          (declare local (VarDecl ;)+) ?
                          do (Instruction)+ done
   ```

Un programme (1) ou (2) est bien *typé* si ses différents composants (déclarations et instructions) sont eux-mêmes bien typés et si l'ExprD des clauses **when** est bien typée et booléenne. La *sémantique* d'un programme $PILS$ est donnée par une fonction qui, pour un flux d'entrée donné ($i \in \mathbb{I}$), donne (si l'exécution se passe bien) un ensemble de valeurs entières sur le flux de sortie ($o \in \mathbb{O}$) :

$$[\![.]\!] : \text{Programme} \rightarrow \mathbb{I} \nrightarrow \mathbb{O}$$

Intuitivement, la sémantique d'un programme $PILS$ est différente de celle que l'on peut trouver dans un langage de programmation classique. En effet, le langage $PILS$ est un DSL, utilisé pour décrire le comportement d'un EA sur une grille de jeu en fonction de l'environnement qui l'entoure. Un programme $PILS$ sera donc exécuté complètement *à chaque tour de jeu*. Le jeu communiquera avec le programme $PILS$ au travers des entrées (i) et sorties (o) de ce programme, définies au niveau d'un état $s = (\sigma, \epsilon, i, o)$. Afin de permettre la définition de stratégies plus fines, le jeu s'assurera de conserver la valeur des différentes *variables globales* de tour en tour (concrètement, le jeu fournira en entrée du programme les valeurs de ces variables au début de son exécution). L'on découpera l'exécution d'un programme en 3 phases :

1. la création d'un contexte vide dans lequel seront ajoutées en début d'exécution les variables d'environnement, ainsi que les déclarations de variables globales. Ces variables seront initialisées sur base des valeurs fournies par le jeu sur le flux d'entrée du programme $PILS$ (nous donnons l'ordre dans lequel le jeu fournira ces valeurs un peu plus loin).
2. L'exécution du programme $PILS$ jusqu'à ce qu'une instruction **next** Action soit rencontrée.
3. L'écriture sur le flux de sortie de la *valeur associée à l'action* figurant dans l'instruction **next** Action (telle que définie pour chaque action à la section 9.2), ainsi que les valeurs des variables globales dans l'ordre dans lequel celles-ci sont déclarées.

Afin que ceci soit possible (en particulier lors de la première exécution du programme $PILS$ où les variables globales n'ont pas encore été initialisées), l'on considèrera que le flux d'entrée i comme la concaténation des valeurs fournies par le jeu et d'une séquence infinie de 0.

Comme pour les fonctions, nous simplifions la notation pour décomposer un programme de la manière suivante : *init dvg dfg cws exit*, où :

— *init* est la phase d'initialisation du programme ;
— *dvg* est la liste des déclarations de variables globales (dans leur ordre de déclaration) ;
— *dfg* est la liste des déclarations de fonctions ;
— *cws* est la liste des clauses **when** (en ce compris la clause par défaut) ;
— *exit* est la phase de terminaison du programme.

12.1 Création de l'état initial du programme

Lors du lancement du programme l'état initial $s = (\sigma, \epsilon, i, o)$, où le flux d'entrée i a été initialisé par le jeu appelant le programme $PILS$, le flux de sortie o est vide, l'environnement ϵ ne contient qu'un seul contexte (qui est vide) et la mémoire σ est vide. La phase d'initialisation est définie de la manière suivante :

$$[\![init\ dvg\ dfg\ cws\ exit]\!](\sigma, \epsilon, i, o) = [\![dvg\ dfg\ cws\ exit]\!](\sigma_n, \epsilon_n, i_n, o_n)$$

Où, pour un flux d'entrée $i = (v_1, \ldots, v_k, 0, \ldots, 0, \ldots)$, on défini $s_n = (\sigma_n, \epsilon_n, i_n, o_n)$ comme :

$$\begin{array}{lcl}
(\sigma_1, \epsilon_1, i_1, o_1) & = & (\sigma_1, \epsilon[\textbf{latitude}/v_1], (v_2, \ldots), o), \text{ où } (a, \sigma_1) = Alloc(Int)\sigma \\
(\sigma_2, \epsilon_2, i_2, o_2) & = & (\sigma_2, \epsilon_1[\textbf{longitude}/v_2], (v_3, \ldots), o_1), \text{ où } (a, \sigma_2) = Alloc(Int)\sigma_1 \\
(\sigma_3, \epsilon_3, i_3, o_3) & = & (\sigma_3, \epsilon_2[\textbf{grid size}/v_3], (v_4, \ldots), o_2), \text{ où } (a, \sigma_3) = Alloc(Int)\sigma_2 \\
\ldots & \ldots & \ldots
\end{array}$$

En d'autres termes, la phase d'initialisation se chargera de définir et d'initialiser la valeur des différentes variables d'environnement sur base des valeurs fournies sur le flux d'entrée. Les valeurs seront fournies dans l'ordre suivante :

1. **latitude**
2. **longitude**
3. **grid size**
4. **map count**

5. **radio count**
6. **ammo count**
7. **fruits count**
8. **soda count**
9. **life**
10. **ennemi is north** (valeur 0 pour *faux* et 1 pour *vrai*)
11. **ennemi is east** (valeur 0 pour *faux* et 1 pour *vrai*)
12. **ennemi is south** (valeur 0 pour *faux* et 1 pour *vrai*)
13. **ennemi is west** (valeur 0 pour *faux* et 1 pour *vrai*)
14. **graal is north** (valeur 0 pour *faux* et 1 pour *vrai*)
15. **graal is east** (valeur 0 pour *faux* et 1 pour *vrai*)
16. **graal is south** (valeur 0 pour *faux* et 1 pour *vrai*)
17. **graal is west** (valeur 0 pour *faux* et 1 pour *vrai*)
18. **nearby[0 , 0]** (la valeur correspond au type de case, comme spécifié à la Section 7.3.2)
19. **nearby[0 , 1]** (la valeur correspond au type de case, comme spécifié à la Section 7.3.2)
20. **nearby[0 , 2]** (la valeur correspond au type de case, comme spécifié à la Section 7.3.2)

...

99 **nearby[8 , 8]** (la valeur correspond au type de case, comme spécifié à la Section 7.3.2)

12.2 Initialisation des variables globales

Une fois l'état initial créé et les variables d'environnement initialisées, l'on peut définir les variables globales et les initialiser avec les valeurs fournies sur le flux d'entrée. Dans la pratique, ces valeurs sont fournies au programme $PILS$ par le jeu, qui se chargera des les préserver d'un tour à l'autre. La définition et l'initialisation des variables globales se fait de la manière suivante :

$$[\![dvg\ dfg\ cws\ exit]\!]s = [\![dfg\ cws\ exit]\!][\![dvg]\!]s$$

Où $[\![dvg]\!]s$ met à jour l'environnement en déclarant et initialisant les variables globales. À nouveau, la définition de $[\![dvg]\!]s$ est inductive : $[\![dvg]\!]s = [\![t_1 var_1, t_2 var_2, \ldots, t_n var_n]\!](\sigma, \epsilon, (v_{100}, v_{101}, \ldots), o) = (\sigma_n, \epsilon_n, i_n, o_n)$. Les 99 premières valeurs sur le flux d'entrée ayant déjà été consommées lors de l'initialisation des variables globales. L'on a donc :

$$\begin{array}{rcl} (\sigma_1, \epsilon_1, i_1, o_1) & = & (\sigma_1, \epsilon[var_1/v_95], (v_{100}, \ldots), o), \text{ où } (a, \sigma_1) = Alloc(t_1)\sigma \\ (\sigma_2, \epsilon_2, i_2, o_2) & = & (\sigma_2, \epsilon_1[var_2/v_96], (v_{101}, \ldots), o_1), \text{ où } (a, \sigma_2) = Alloc(t_2)\sigma_1 \\ \ldots & \ldots & \ldots \end{array}$$

Dans le cas d'une variable tableau, les valeurs sont fournies sur le flux d'entrée de la même manière que la variable d'environnement **nearby**, l'initialisation d'un tableau à k cases demandera donc de consommer k entiers sur le flux d'entrée.

12.3 Déclaration des fonctions

Les fonctions sont initialisée suivant la définition sémantique fournie à la Section 10. L'on a donc simplement :

$$[\![dfg\ cws\ exit]\!]s = [\![cws\ exit]\!][\![dfg]\!]s$$

12.4 Traitement des clauses when

Les clauses **when** sont mutuellement exclusives en $PILS$. Intuitivement, la première clause dont la garde est satisfaite est exécutée et toutes les clauses suivantes sont ignorées. Le résultat de l'exécution d'une clause **when** sera soit une valeur correspondant à la prochaine action à effectuer, spécifiée par une instruction **next** Action (cf. Section 9), soit une valeur indéfinie (notée $\perp$) si aucune instruction **next**

Action n'a été exécutée dans la clause **when**. Une clause **when** peut elle même être décomposée en 3 éléments : une garde (g), une liste de déclarations de variables locales (dvl), une suite d'instruction (il) à exécuter si la garde est vraie. L'on considèrera que la garde de la clause ClauseDefault est toujours vraie. Formellement, l'on a donc :

$$[\![((g_1, dvl_1, il_1), \ldots, (g_n, dvl_n, il_n))\ exit]\!]s = \begin{cases} [\![exit]\!][\![il_1]\!][\![dvl_1]\!]s_1 \\ \quad \text{si } [\![g_1]\!]s = (vrai, s_1) \\ [\![((g_2, dvl_2, il_2), \ldots, (g_n, dvl_n, il_n))\ exit]\!]s_1 \\ \quad \text{si } [\![g_1]\!]s = (faux, s_1) \end{cases}$$

L'on remarquera ici que l'évaluation d'une garde a de possibles effets de bord qui peuvent se répercuter sur l'évaluation des gardes suivantes (ceci est dû à la possibilité d'appeler des fonctions depuis une garde).

12.5 Terminaison du programme

Une fois la clause **when** exécutée (et donc la prochaine action à mener potentiellement déterminée), il reste deux choses à faire : notifier le jeu de la prochaine action à effectuer en imprimant la valeur correspondant à cette action sur la sortie standard ; et imprimer sur la sortie standard la valeur des différentes variables globales afin que le jeu puisse les (re)fournir en entrée lors de la prochaine exécution du programme $PILS$ au tour suivant. Toutes ces valeurs sont imprimées sur le flux de sortie, en commençant par la prochaine action à effectuer suivi des valeurs des variables globales. L'exécution d'une clause **when** retourne un résultat (une valeur et un état) avec la valeur de la prochaine action à effectuer (n) ou $\perp$ si aucune valeur n'a été spécifiée. Dans ce dernier cas, l'on considèrera que l'action par défaut est **do nothing**.

$$[\![exit]\!](\perp, s) = [\![exit]\!]([\![\textbf{next do nothing}]\!], s)$$

$$[\![exit]\!](n, (\sigma, \epsilon, i, o)) = (\sigma, \epsilon, i, o \cdot n \cdot v_1 \cdot \ldots \cdot v_n)$$

Où $v_1, \ldots, v_n$ correspondent aux valeurs des variables globales $var_1, \ldots, var_n$, calculées en utilisant $[\![var_1]\!]s, \ldots, [\![var_n]\!]$. En cas de tableau à plusieurs dimensions, nous considèrerons que chaque case est imprimée sur le flux de sortie (dans le même ordre que pour l'initialisation à partir du flux d'entrée).

13 Commentaires

Les commentaires en $PILS$ se placent entre deux délimiteurs comme spécifié ci-dessous :

Commentaires ::= /* (.*) */

La sémantique d'un commentaire retourne simplement l'état courant (les commentaires étant ignorés lors de l'exécution) : $[\![\text{Commentaires}]\!]s = s$.

14 Exemple de stratégie $PILS$

```
declare and retain

        import inputFile.wld
        t as boolean;
        res as integer;
        f as function(): boolean
        do
                set t to true
                /* other instructions here */
        return t;
        done
        facto as function(x as integer):integer
        do
                if x < 0 or x = 0 then
                        set res to 1
                else
```

```
                        set res to x * facto(x - 1)
                done
         return res;
                done
when your turn
        when life < 20 do
                next use soda
        done
        when life < 50 do
                next use fruits
        done
        when nearby[4,5] = ennemi do
                next shoot north
        done
        by default /* Default */
        declare local
                b as boolean;
        do
                /* b is false by default */
                set b to f()
                /* b is now true */
                next move east
        done
```

Bon Travail !