TD7 — Tests de programmes robots

1. Programmes robots

Le robot Curiosity évolue maintenant sur un terrain de manière autonome, à partir d'un *programme*. Ce programme est une séquence de caractères (`'A'`, `'G'`, `'D'`, `'M'`, etc.), lue dans un fichier. Un paquetage d'*interprétation* d'un programme est fourni. Le but de cette deuxième semaine est de tester de manière systématique cet interprète.

La syntaxe complète du langage de programmation du robot Curiosity est rappelée en annexe.

2. Paquetage `environnement`

L'*environnement* est la composition d'un *terrain* et d'un *robot*. C'est sur cet environnement que vont être effectuées les actions de base du robot : *avancer*, *tourner à gauche*, *tourner à droite* et *effectuer une mesure*. Ces actions seront réalisées par l'interprète du programme.

Voici ci-dessous un extrait de ce paquetage :

```
/* Initialise l'environnement envt :
   - lit le terrain dans le fichier fichier_terrain
   - initialise le robot : coordonnées initiales lues dans le fichier
   terrain, orientation initiale vers l'est
*/
erreur_terrain initialise_environnement(Environnement *envt,
                                        char *fichier_terrain);
/* Afficher le terrain avec la position et l'orientation du robot */
void afficher_envt(Environnement *envt);
```

3. Paquetage `programme`

Le paquetage `Programme` fournit le type de données et la fonction de lecture d'un programme dans un fichier :

```
/* Programme : séquence de commandes */
typedef struct {
  Commande tab[PROG_TAILLE_MAX];
  int lg;
} Programme;

/* Erreurs de lecture d'un programme */
typedef enum {
  OK_PROGRAMME,
  ERREUR_FICHIER_PROGRAMME,
  ERREUR_BLOC_NON_FERME,
  ERREUR_FERMETURE_BLOC_EXCEDENTAIRE,
  ERREUR_COMMANDE_INCORRECTE
} type_erreur_programme;

typedef struct {
  type_erreur_programme type_err;
  char *ligne; /* Ligne du programme contenant l'erreur */
  int num_ligne, num_colonne; /* Position de l'erreur dans le fichier */
} erreur_programme;

/* Lecture d'un programme prog dans le fichier nom_fichier */
erreur_programme lire_programme(Programme *prog, char *nom_fichier);
```

4. Paquetage `interprete`

Le paquetage `interprete` fournit les fonctions d'interprétation d'un programme : initialisation d'un état, exécution d'un pas du programme.

```
1  /* Résultat de l'interprétation */
2  typedef enum {
3    OK_ROBOT, /* Le robot est sur une case libre et le programme n'est pas terminé
4              */
5    SORTIE_ROBOT, /* Le robot est sorti du terrain */
6    ARRET_ROBOT, /* Le programme est terminé */
7    PLOUF_ROBOT, /* Le robot est tombé dans l'eau */
8    CRASH_ROBOT, /* Le robot est rentré dans un rocher */
9    ERREUR_PILE_VIDE, /* Erreur : pile vide */
10   ERREUR_ADRESSAGE, /* Erreur d'adressage : indice de commande incorrect */
11   ERREUR_DIVISION_PAR_ZERO, /* Erreur : tentative de division par 0 */
12 } resultat_inter;
13
14 /* Etat de l'interprè te */
15 typedef struct {
16   /* Program counter : adresse de la prochaine commande àexécuter */
17   int pc;
18   /* Pile de données */
19   PileEntiers stack;
20   /* Pile d'adresses de retour */
21   PileEntiers sp;
22 } etat_inter;
23
24 /* Initialisation de l'état */
25 void init_etat(etat_inter *etat);
26
27 /* Pas d'exécution de l'interprè te : exécute une commande, modifie
28    l'environnement et l'état, renvoie l'état du robot */
29 resultat_inter exec_pas(Programme *prog, Environnement *envt, etat_inter *etat);
```

Exercice 1. Liens de dépendances

Donner les liens de dépendances entre ces différents paquetages. Écrire leurs règles Makefile de compilation.

Exercice 2. Exécution simple

À l'aide de ces paquetages, écrire un programme prenant en argument deux noms de fichiers (un terrain et un programme), et qui exécute ce programme sur un robot dans ce terrain.

Donner la règle Makefile pour la compilation et l'édition de liens de ce programme.

Exercice 3. Tests fonctionnels

Décrire un protocole complet permettant de tester l'interprète.

Exercice 4. Tests de robustesse

Donner des exemples de programmes déclenchant une erreur :

— à la lecture du programme ;

— à l'exécution du programme.

A. Langage de programmation du robot Curiosity

Rappel : la syntaxe [$a_1 \quad a_2 \cdots a_n \quad$ `C`] signifie que la commande `C` prend n arguments, récupérés sur la pile (le dernier argument a_n est en haut de la pile). Dans tous les cas, les arguments sont enlevés de la pile avant l'exécution de la commande.

Commande — **Sémantique et exemples**

`A` — Avancer le robot d'une case

`G` — Tourner le robot de 90° vers la gauche

`D` — Tourner le robot de 90° vers la droite

n **`M`** — Effectuer une «mesure» :

Valeurs d'entrées		Valeurs de sortie	
0	sur place	0	rien
1	devant	1	eau
2	devant droite	2	rocher
3	droite		
4	derrière droite		
5	derrière		
6	derrière gauche		
7	gauche		
8	devant gauche		

`{` *commandes* **`}`** — Crée un groupe de commandes *{commandes}* et le place sur la pile

cmd **`!`** — Exécute *cmd*

Exemple :

Pile	Programme
vide	`{ A D } { G A } ! !`
`{ A D }`	`{ G A } ! !`
`{ A D } { G A }`	`! !`
`{ A D }`	`G A !`
`{ A D }`	`!`
vide	`A D`
vide	*vide*

Dans cet exemple, Curiosity va d'abord tourner à gauche, puis avancer deux fois, puis tourner à droite.

n V F **`?`** — Instruction conditionnelle : exécute V si $n \neq 0$, et F si $n = 0$.

Exemple :

Pile	Programme
vide	`3 M { G A } { A } 7 M { D A } { A } ? ?`
`0`	`{ G A } { A } 7 M { D A } { A } ? ?`
`0 { G A } { A }`	`7 M { D A } { A } ? ?`
`0 { G A } { A } 2 { D A } { A }`	`? ?`
`0 { G A } { A }`	`D A ?`
`0 { G A } { A }`	`?`
vide	`A`
vide	*vide*

Ce code fait d'abord tourner à droite et avancer Curiosity, puis le fait avancer une nouvelle fois.

A B **`X`** — Échange A et B sur la pile

x y op — avec $op \in \{+, -, *, /\}$: calcule *x op y* et place le résultat sur la pile

$A_n \ldots A_2\ A_1\ n\ x$ **`R`** — Effectue une « rotation » de x pas vers la gauche des n éléments en haut de la pile : le haut de la pile devient $A_{n-x} \ldots A_1\ A_n\ A_{n-1} \ldots A_{n-x+1}$

e **`C`** — Clone *e* sur la pile

cmd n **`B`** — Exécute *cmd*, décrémente n sans enlever `B` du programme si $n > 0$. Sinon enlève *cmd* et n de la pile, et `B` du programme.

e **`I`** — Ignore l'élément en haut de la pile