C Learning Environment

RS : Réseaux et Systèmes

Deuxième année

L'objectif de ce projet est de réaliser une version simplifiée de *Java Learning Machine* en C.

1 Logistique

1.1 Comment travailler

Apprendre à travailler en groupe fait partie des objectifs pédagogiques de ce projet. Il vous est donc demandé de travailler en binôme. Vous pouvez travailler par groupe de trois si vous le souhaitez (mais sachez que nous tiendrons compte de la taille des groupes lors de la notation – regardez du côté des extensions pour compenser). Il est **interdit** de travailler seul. Un ingénieur travaille rarement seul.

La suite du sujet contient une stratégie possible que vous êtes libre de suivre, ou non.

Il est **indispensable** que votre projet fonctionne (aussi) sous linux.

1.2 Évaluation de ce projet

Rapport. Vous devez rendre un mini-rapport de projet (5 pages maximum hors annexes et page de garde, format pdf). Vous y détaillerez les difficultés auxquelles vous avez été confronté, et comment vous les avez résolues. Vous indiquerez également le nombre d'heures passées sur les différentes étapes de ce projet (conception, codage, tests, rédaction du rapport) par membre du groupe.

Soutenances Vous devrez nous faire une démonstration de votre projet et être prêts à répondre à toutes les questions techniques sur le codage de l'application (sans doute à la mi-janvier).

Section «Remerciements» du rapport. Votre rapport doit contenir quelque chose comme la mention suivante : «Nous avons réalisé ce projet sans aucune forme d'aide extérieure» ou bien «Nous avons réalisé ce projet en nous aidant des sites webs suivants (avec une liste de sites, et les informations que vous avez obtenu de chaque endroit)» ou encore «Nous avons réalisé ce projet avec l'aide de (nommer les personnes qui vous ont aidé, et indiquez ce qu'ils ont fait pour vous)». Si la liste est trop longue, vous pouvez la déplacer en annexes (pour ne pas empiéter sur vos 5 pages de rapports). Rien ne vous empêche de vous faire aider, mais il vous est demandé un minimum d'honnêteté académique en listant vos aides. Tout manquement à cette règle sera sanctionné comme il se doit.

La tricherie sera **sévèrement punie**[1]. Par se faire aider, on entend : avoir une discussion sur le design du code, y compris sur des détails techniques et/ou les structures de données à mettre en œuvre. Ceci est fortement encouragé. Par tricher, on entend : copier ou recopier du code ou encore lire le code de quelqu'un d'autre pour s'en inspirer. Nous testerons l'originalité de votre travail par rapport aux autres projets rendus[2], et il est difficile de défendre en soutenance un projet que l'on n'a pas écrit entièrement.

1.3 Rendre votre projet

Que rendre ? Il est attendu que vous rendiez le rapport, vos sources, un Makefile et un fichier AUTHORS listant les membres du groupe (un login unix par ligne). Le script ci-dessous se charge de copier tout ces fichiers, vous n'avez pas à constituer d'archive.

Comment rendre votre copie Vous devez placer les fichiers nécessaires dans un répertoire sur neptune, vous placer dedans, et invoquer la commande suivante (seuls les fichiers utiles sont copiés). Vous pouvez invoquer cette commande autant de fois que vous le souhaitez. À chaque invocation, les fichiers précédemment déposés sont effacés et remplacés avec les nouveaux.

`/home/EqPedag/quinson/bin/rendre_projet RS`

Avant le vendredi 7 janvier, à 12h.

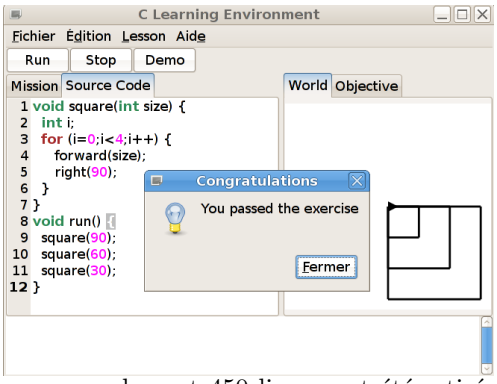
(le script n'acceptera pas de soumission en retard)

Voir `http://www.loria.fr/~quinson/teach-RS.html` d'éventuelles informations complémentaires.

1. cf. `http://www.loria.fr/~quinson/teaching.html`
2. cf. `http://theory.stanford.edu/~aiken/moss/`

2 Description de CLE (C Learning Environment)

Le projet de cette année vise à constituer un environnement d'apprentissage du C proche de ce que la JLM (Java Learning Machine) offre en Java. Rassurez vous, vous n'avez pas à réaliser toutes les fonctionnalités de la JLM. Mais vous serez amenés à utiliser quasiment toutes les notions vues dans le cours, et ce thème offre de très nombreuses extensions possibles. Enfin, il s'agit sans doute de votre premier programme vraiment utile, puisque le résultat de ce projet sera réutilisé en 1A dans le module de CSH dès cette année.



Ne vous laissez cependant pas impressionner outre mesure : seulement 450 lignes ont été retirées du template fourni par rapport à l'implémentation de démonstration (qui, il est vrai, n'implémente aucune extension proposée). Il ne s'agit donc pas d'un projet si gros que cela (juste un peu technique).

3 Indications pour réaliser votre projet

3.1 Outils à utiliser

Les fichiers sources de l'interface (basée sur GNOME) vous sont fournis. Vous êtes libre de ne pas les utiliser et refaire votre propre interface, mais gardez à l'esprit qu'il s'agit d'un projet de système, pas d'IHM. Vous pouvez bien entendu modifier les sources fournis pour les adapter à vos goûts esthétiques. Si vous souhaitez utiliser un autre environnement que GNOME, veuillez en référer à votre encadrant au préalable. En particulier, KDE n'est pas une bonne idée dans notre cadre puisqu'il est impossible de l'utiliser depuis le C.

Pour pouvoir compiler le source fourni, il faut installer un certain nombre de paquets de développement. Il vous faut au minimum `libgtksourceview2.0-dev` et `libglade2-dev`, mais je n'ai malheureusement pas la liste exacte. Lisez les éventuels messages d'erreur pour trouver ce qu'il manque.

Vous serez peut-être amenés à consulter la documentation de GNOME. Celle disponible sur les sites web est difficile à naviguer car il y a des liens cassés. La meilleure solution est d'utiliser Anjuta pour lire cette documentation.

Vous serez amenés à utiliser la plupart des appels systèmes vus en cours, et plus de détails vous seront donnés en temps et en heure dans la stratégie ci-dessous.

Pour exécuter les tortues au ralenti et permettre à l'utilisateur de profiter des animations, il est nécessaire d'utiliser des threads. Le plus simple est d'utiliser l'interface `g_thread_*` fournie par la Glib au cœur de GNOME. Cette interface est très proche des pthreads vus en cours.

3.2 Stratégie possible

3.2.1 Comprendre le template de projet fourni

Vous trouverez à l'adresse `http://www.loria.fr/~quinson/teach-prog.html` une archive contenant une première version du projet à réaliser. Cela vous permettra de vous concentrer sur les aspects système de ce projet. Vous pouvez cependant adapter l'interface fournie si vous le souhaitez. Lorsque vous ajouterez les fonctionnalités demandées, il est préférable que vous placiez vos fonctions dans les fichiers existants en respectant le découpage actuel.

L'interface graphique. Voici les fichiers principaux de l'interface fournie :

- `UI/CLE.glade` : description de l'interface, modifiable avec le programme `glade`. Voir le tutoriel suivant si vous le souhaitez : `http://live.gnome.org/Glade/Tutorials`
- `UI/CLE.c` : contient la fonction main de l'application. Elle charge le fichier glade, initialise ce qui doit l'être, puis lance la boucle principale de GNOME. Ce fichier contient également quelques fonctions utiles que vous n'avez normalement pas à modifier. Voir `UI/CML.h` pour leur interface.
- `UI/callbacks.c` : contient les fonctions invoquées quand on clique sur les boutons de l'interface graphique (Run, Stop et Demo). Dans GNOME, ces fonctions sont nommées signaux, mais cela n'a rien à voir avec les signaux POSIX vus en cours. Le bouton *Demo* est déjà implémenté et vous

n'avez normalement pas à le modifier. Vous pouvez vous en inspirer pour implémenter les autres boutons.

Le modèle de données des exercices. Plusieurs fichiers sont utilisés :

- `core/exercise.c` et `core/exercise.h` définissent le type de données d'un exercice en particulier.
- `core/lesson.c` et `core/lesson.h` définissent le type de données d'une leçon (*i.e.*, d'un ensemble cohérent d'exercices).
- `lessons/logo_threesquare.c` et `lessons/logo_circle.c` sont deux exemples d'exercices. Le premier consiste simplement à faire dessiner trois carrés à l'écran tandis que le second demande de dessiner un cercle. D'autres exercices sont également fournis.

Comme sa grande sœur JLM, le CLE offre plusieurs exercices à l'utilisateur. Pour changer d'exercice dans l'interface, il faut invoquer la méthode `lesson_set_exo()` (définie dans core/lesson.h), qui prend en argument une leçon et le numéro dans la liste de l'exercice à charger. Les leçons sont créées par les fonctions `lesson_main()` définies dans lessons/logo.c ou lessons/recursion.c [3] à partir d'instances d'exercices. Ces instances sont créées par les fonctions principales des différents fichiers `lessons/logo_*.c`. Pour créer une instance d'exercice, ces fonctions utilisent `exercise_new()` (définie dans `core/exercise.h`). Cette fonction prend 4 arguments : deux chaînes décrivent l'exercice aux élèves (l'une est le texte explicatif de la mission et l'autre est le texte initial du code), un pointeur sur une fonction permettant de calculer l'état visé du monde (c'est ce que font les tortues de la démo) et un monde initial.

Pour le bon fonctionnement de CLE, trois mondes différents doivent être définis pour chaque exercice : le monde initial (`exo->w_init`), le monde courant (`exo->w_curr`) et le monde objectif (`exo->w_goal`). Le deuxième et le troisième sont visibles depuis l'interface graphique, respectivement dans les onglets *World* et *Objective*. Le monde passé en argument à `exercise_new` est utilisé pour remplir `w_init` ; ce champ ne sera jamais modifié avant la destruction de l'objet exercice. `w_curr` est le monde qui doit être modifié quand le code des élèves s'exécute. À l'initialisation et à chaque fois que le bouton *Reset* de l'interface est cliqué, `w_curr` est réinitialisé comme copie conforme de `w_init`. Le troisième monde, `w_goal`, est l'état attendu du monde à la fin de l'exécution du code des élèves. Le mode *Demo* fonctionne de manière similaire en réinitialisant `w_goal` à l'état de `w_init`, puis en exécutant le code de la fonction passée en troisième argument à `exercise_new()`.

L'exécution de la démo est donc implémentée dans le code fourni, et vous pouvez vous en inspirer pour écrire celle du code des élèves. En particulier, ce code est multi-threadé pour que l'interface continue à s'afficher correctement pendant les déplacements des tortues (cf. §3.2.4).

Le modèle de données du monde et des tortues. L'état de chaque monde est représenté par deux types de données : les tortues, et le monde à proprement parler. Bien qu'il s'agisse de code C, nous suivons l'approche objet pour les définir. En particulier, toutes les fonctions prennent comme premier argument l'objet sur lequel elles doivent agir. Par exemple, `turtle_forward(t, 12)` serait écrite de la façon suivante en Java : t.forward(12).

Pour chaque type de données, vous trouverez dans le fichier d'entête (`logo/turtle.h` et `logo/world.h`) les fonctions associées. Leur implémentation se trouve naturellement dans les fichiers `logo/turtle.c` et `logo/world.c`. Pour chaque type de données, cela commence par les fonctions de gestion de la mémoire suivantes :

- *class*`_new()` : constructeur (chargé de faire un malloc et initialiser l'instance).
- *class*`_free(i)` : destructeur (chargé de rendre la mémoire prise par l'instance).
- *class*`_eq(i1,i2)` : comparateur (renvoie un booléen indiquant si les deux instances passées en paramètres sont équivalentes)
- *class*`_copy(i)` : copy constructeur (chargé de créer une nouvelle instance étant la copie à l'identique de celle passée en argument).

Après ces fonctions de gestion de la mémoire viennent les fonctions de l'interface de la classe. Dans le cas des tortues, il s'agit des six fonctions de l'interface utilisateur (`forward()`, `backward()`, `turnLeft()`, `turnRight()`, `penUp()` et `penDown()`). Dans le cas du monde, il s'agit de `line_add()`, qui permet aux tortues d'ajouter un trait à l'ensemble de ceux déjà dessinés sur le monde, et `redraw()` qui redessine le monde dans l'interface graphique. Notez que `world_line_add()` utilise un mutex pour être *thread-safe* et permettre à plusieurs tortues de l'invoquer en même temps. La fonction `world_redraw()` ne doit pas être invoquée directement, mais au travers de l'interface graphique. Pour provoquer son appel, vous devez utiliser `world_ask_repaint()`, définie dans `UI/CLE.c`.

3. Il est normal que ces deux fichiers déclarent tous deux une fonction du même nom "lesson_main()" – cf. §3.2.7.

Ces deux classes sont bien entendu liées : le monde a besoin de connaître les tortues qui le peuplent pour implémenter la démo et autres tandis que la tortue a besoin de connaître le monde dans lequel elle évolue pour savoir où ajouter les traits qu'elle trace. `turtle_set_world()` permet d'indiquer son monde à une tortue tandis que `world_turtle_add()` permet d'ajouter une tortue dans le monde. Le copy constructor du monde invoque les copy constructors des tortues et réinscrit les tortues dans le bon monde.

Remarque pour les perfectionnistes : Dans une architecture orientée objet et MVC, les fonctions attachées à la vue ne devraient pas être ainsi déclarées aux côtés de celles attachées au modèle de données. Cependant, la fonction `world_redraw()` n'a aucun état persistant puisque le descripteur de surface sur laquelle dessiner est passé en argument (`cairo_t *cr`). Il s'agirait donc d'une méthode de classe et notre entorse aux canons de design d'interface nous semble tolérable. Si ce n'est pas votre cas, libre à vous d'améliorer les choses. Les prototypes de ces fonctions sont cependant placées dans un fichier d'entête séparé (`world_view.h`), entre autre pour simplifier la compilation des plugins (sans les paquets *-dev).

3.2.2 Compilation

La première étape est de compiler le code de l'élève quand il clique sur *Run*. Pour cela, créez une méthode `exercise_run` prenant l'exercice et le code à exécuter en argument, puis invoquez la depuis le bon callback.

Mais comment faire en sorte d'exécuter le code des élèves dans votre programme ? Une idée serait de le compiler sous forme de greffon (*plugin*) puis de le charger dans l'espace mémoire de CLE afin de l'exécuter. Cette approche présente un gros défaut : si le code de l'élève provoque une erreur de segmentation, cela tuera l'interface graphique. Nous allons donc compiler le code des élèves comme une application indépendante (§3.2.3). Ensuite, nous exécuterons ce code dans un processus séparé pour plus de sécurité (§3.2.4), puis nous gérerons plus tard (§3.2.5) les communications entre les processus.

Dans la fonction `exercise_run()`, il faut écrire le code à exécuter dans un fichier (cf. mkstemp(2) pour choisir un nom de fichier original à chaque fois), puis le compiler. Pour cela, il faut exécuter gcc dans un processus fils après avoir pris soin de créer un tube pour que le père puisse lire la sortie d'erreur du fils. Si le nom de fichier source ne termine pas par ".c" (comme c'est le cas si vous utilisez à raison mkstemp), il faut ajouter "-x c" aux arguments de gcc pour lui préciser qu'il s'agit de source C. Utilisez la fonction `CLE_log_append()` pour afficher la sortie d'erreur de gcc. Attention, cette fonction `append` invoque `free()` sur la chaîne passée en paramètre (pour des raisons de stabilité de stockage dus au threads), et il faut donc utiliser strdup() pour dupliquer ce que vous lui donnez. À cette étape, il est normal que le code fourni ne compile pas à cette étape puisqu'il utilise par exemple `forward()` sans le définir. Ne laissez pas le processus `gcc` devenir un zombie.

3.2.3 Faire un binaire du code des élèves

Il s'agit maintenant de définir les fonctions que les étudiants pourront invoquer dans leur code. Pour savoir lesquelles définir, consultez logo/turtle.h. Vous devez par exemple définir `double get_x()`, `void left(int angle)` ou `void forward(double steps)`. Placez ces fonctions dans `logo/turtle_userside.c`. En guise de corps de fonction, placez simplement des `printf`s afin de suivre les appels de fonction, comme dans l'exemple ci-dessus.

```
void forward(double steps) {
  printf("FORWARD %d\n",steps);
}
```

Faites en sorte que le fichier que vous venez d'écrire soit compilé avec le code des étudiants. Pour éviter de forcer les étudiants à inclure explicitement un fichier d'entête, vous pouvez écrire les prototypes des différentes fonctions dans le fichier temporaire où vous écrivez le code à compiler. Le plus simple est d'injecter directement les entêtes au début du fichier contenant le code de l'élève sans passer par un `#include`. Le problème est évidement que cela décale les numéros de ligne dans les messages d'erreur de gcc. La solution est d'ajouter la ligne `#line 1 "yourcode"` juste avant de mettre le code de l'élève dans le fichier à compiler. Cette primitive indique à gcc qu'il doit considérer se trouver à la première ligne du fichier `yourcode` à partir de maintenant.

Ajoutez ensuite une fonction `main()` dans `logo/turtle_userside.c`. Faites lui simplement exécuter la fonction `run()` définie par les élèves.

La compilation devrait maintenant bien se dérouler, et vous devriez obtenir un programme exécutable quelque part dans `/tmp` quand vous cliquez sur *Run*. Quand j'exécute le mien manuellement, j'obtiens :

```
FORWARD 90.000000
RIGHT 90.000000
FORWARD 90.000000
RIGHT 90.000000
FORWARD 90.000000
RIGHT 90.000000
```

```
FORWARD 90.000000
RIGHT 90.000000
FORWARD 60.000000
RIGHT 90.000000
FORWARD 60.000000
RIGHT 90.000000
FORWARD 60.000000
RIGHT 90.000000
FORWARD 60.000000
RIGHT 90.000000
FORWARD 30.000000
RIGHT 90.000000
FORWARD 30.000000
RIGHT 90.000000
FORWARD 30.000000
RIGHT 90.000000
FORWARD 30.000000
RIGHT 90.000000
```

3.2.4 Exécuter le code des élèves

Il faut maintenant dupliquer le code en charge d'exécuter la démo pour écrire le code en charge d'exécuter le code des élèves. Il s'agit surtout de préparer le fait que chaque tortue du code des élèves s'exécute dans un thread séparé, et le fait que ces threads soient pilotés par un autre thread. Réalisez cette étape de duplication avec beaucoup d'attention car si des morceaux de l'exécution du code utilisent des fonctions ou des variables globales de l'exécution de la démo, vous allez au devant de gros problèmes.

Pour ce qui est de l'exécution du code à proprement parler, il faut changer le fait que `turtle_run` exécute `exo->solution`. À la place, il faut lui faire exécuter le code compilé à l'étape précédente. Pour cela, on nous allons définir une méthode `exercise_run_one(turtle_t)` réalisant ces opérations, et utiliser `turtle_set_code()` pour faire en sorte que toutes les tortues du monde `w_curr` exécutent cette fonction.

Avant de vous lancer dans le code de `exercise_run_one()` à proprement parler, placez des `printf()`s dans votre code pour vérifier le bon fonctionnement de cette étape.

```
Launch all turtles
Run turtle 0xa00c0c8 (actually, sleeping)
Not restarting the execution (already running)
Done running turtle 0xa00c0c8
End of execution
```

Ajoutez simplement un `sleep(1)` dans le code de `exercise_run_one()`. En cliquant rapidement sur le bouton *Run*, j'obtiens l'affichage ci-dessus. J'ai donc la garantie que ma fonction `exercise_run_one()` est bien invoquée, et que le cas où l'utilisateur tente de relancer l'exécution avant la fin de l'exécution précédente est bien géré.

Il est maintenant temps d'implémenter correctement `exercise_run_one()`. Remplacez le `sleep(1)` par des appels à `pipe()`, `fork()` et `execl()` pour assurer que ce qu'affiche le processus fils arrive correctement dans la console (en utilisant `CLE_log_append()`). Ouvrez également un tube pour contrôler ce que le processus fils peut être amené à lire sur son entrée standard (cela sera utile par la suite). N'oubliez pas de fermer tous les descripteurs inutiles pour pouvoir détecter les fins de flux.

Bien entendu, tout ceci ne doit pas avoir lieu si le processus `gcc` retourne un code de sortie différent de 0 puisque cela dénote d'une erreur de compilation. Dans ce cas, il ne faut pas tenter d'exécuter un binaire qui n'a pas été créé.

On peut ensuite implémenter le fonctionnement du bouton *Stop*. Pour cela, il faut tuer (en envoyant le signal SIGTERM avec `kill()`) tous les processus fils lancés pour exécuter le code des étudiants. Évidement, cela impose de stocker les pid dans une structure partagée, dont les accès devront être correctement protégés par mutex. Dans un premier temps, on peut tricher en remarquant que les exercices proposés ne comportent qu'une seule tortue et donc utiliser une seule variable entière pour stocker tous les pid (mais cette ruse ne fonctionnera plus pour §4.2). On peut aussi utiliser le rang de chaque tortue (donné par `turtle_get_rank()`) pour accéder à la bonne case d'un tableau que l'on créera pour l'occasion (mais cela devra être adapté pour §4.3).

Testez votre code correctement avant de passer à l'étape suivante. Il faut réagir correctement si le code ne compile pas, déclenche une erreur de segmentation ou si l'utilisateur clique sur *Stop* avant la fin de l'exécution, ou encore si l'utilisateur clique sur `Stop` avant le début ou après la fin de l'exécution, etc. On peut vouloir se protéger des cas plus pathologiques comme un clic sur `Stop` en exécution concurrente avec la fin des exécutions et les nettoyages associés.

3.2.5 Faire interagir le code des élèves avec le monde

Il est maintenant temps de faire bouger la tortue du monde quand l'élève appelle la fonction `forward()`. À première vue cela semble difficile puisque le code des élèves s'exécute dans un processus séparé, mais il n'en est rien : le processus `CLE` sait déjà toutes les actions réalisées par les tortues dans les processus

fils. Il affiche même dans la console toutes ces informations. Il suffit donc de changer la boucle qui lit ce que le processus fils écrit pour l'afficher dans la console afin de lui faire interpréter ces ordres.

Pour simplifier cette tâche, on peut commencer par changer les dits affichages en utilisant des codes numériques au lieu de chaînes de caractères. Il s'agit simplement de remplacer le code de la fonction `forward()` injectée au début du code des élèves (cf. §3.2.3) par le code ci-joint. Comme vous le voyez, j'ai choisi de laisser l'affichage en caractères et j'ai *ajouté* l'affichage numérique en début de ligne. Cela peut aider le debogage. Une fois ceci fait, il suffit de changer la boucle en charge de lire l'affichage du processus fils pour recopier dans la console, et interpréter les commandes au fil de l'eau. Le plus simple pour lire le flux ligne à ligne est d'utiliser la fonction `getline()` [4], mais le problème est que cette fonction utilise un `FILE*` et non un *file descriptor*. Qu'à cela ne tienne, un appel à `fdopen` permet de créer un `FILE*` depuis un *file descriptor*. Vous trouverez ci-dessous le détail de la modification faite chez moi pour que cela fonctionne.

```
1  void forward(double steps) {
2    printf("100 %d (forward %d)",
3           steps,steps);
4  }
```

Code avant la modification

```
1  char buff[1024];
2  int got;
3  while ((got = read(c2f[0],&buff,1023))>0) {
4    buff[got] = '\0';
5    CLE_log_append(strdup(buff));
6  }
```

Code après la modification

```
1  FILE *fromchild=fdopen(c2f[0],"r");
2  char *buff=NULL;
3  size_t len=0;
4  int got;
5  while ((got = getline(&buff, &len, fromchild)) != -1) {
6    int cmd=-1;
7    double arg=-1;
8    sscanf(buff, "%d %lf", &cmd,&arg);
9    printf("Seen command %d with argument %lf\n", cmd,arg);
10 }
11 fclose(fromchild);
```

L'étape suivante est de remplacer le `printf` de la ligne 9 par un gros `switch` permettant d'exécuter le bon code (ie, le bon appel à une fonction `turtle_*()` définie dans `turtle.h`) pour répondre à la commande envoyée par le processus fils. Pour répondre aux fonctions qui demandent une valeur de retour (comme `get_x()`), le père devra écrire la réponse dans le tube à destination du fils, et le fils devra lire cette réponse avant de la renvoyer au code de l'élève.

Une fois ce petit changement fait, le code des élèves agit effectivement sur le monde courant. Plaisant, n'est-il pas ?

Il reste cependant un petit problème : si le code des élèves utilise `printf()` (par exemple pour le débug), cela va casser le dialogue de commande entre le processus père et le fils. Pour corriger cela, il faut duppliquer les tubes : deux tubes fils→père (un pour les affichages de l'utilisateur et un pour les commandes) et deux tubes père→fils (l'un pour les retours des commandes, et l'autre n'est pas utilisé pour l'instant. Mais qui sait ? Peut-être que certaines leçons voudront écrire des choses sur l'entrée des codes des élèves ?). Il faut également changer les processus fils et père afin qu'ils utilisent les descripteurs 3 et 4 pour échanger des commandes.

Ensuite, le problème est que le père doit lire de deux sources séparées (tube d'affichage et tube de commande), et que chaque lecture est bloquante par défaut. On pourrait penser à des approches systèmes classiques (comme les I/O non bloquantes ou l'appel système `select()` – certes, c'est hors programme), mais cela n'irait pas forcément car le protocole sur le tube de commande a besoin d'être ligne à ligne. En effet, si on traite la moitié d'une ligne de commande, cela va tout désynchroniser. La meilleure solution est alors de lancer un thread de plus dont le travail est de lire tout ce qui sort du tube d'affichage, et qui l'envoie dans la console. Tous les problèmes disparaissent en 5 à 10 lignes.

3.2.6 Finir les bases

Maintenant que le cœur de l'application fonctionne, il faut finaliser les choses. En particulier, il faut vérifier si le contrat est rempli après l'exécution du code des élèves. Ceci est grandement facilité par la méthode `world_eq()`. La plus grande difficulté est qu'il est interdit en GNOME d'utiliser l'interface graphique depuis un thread autre que le thread principal. Il faut donc utiliser la même ruse que pour l'ajout de texte dans la console en passant avec `g_idle_add()` une fonction que le thread principal exécutera. Consultez le code de la méthode `CLE_log_append()` dans `UI/CLE.c`, et faite de même pour ouvrir la fenêtre de dialogue. Pour créer la fenêtre à proprement parler, il faut utiliser la fonction `gtk_message_dialog_new()`.

Il est maintenant temps de nettoyer votre code, le documenter un peu, et retirer tous les affichages de debug que vous avez placés par ci par là avant de passer à la suite.

4. Cette fonction est spécifique à Linux et n'existe en particulier pas sous Mac OSX. Si vous insistez pour utiliser un Mac, il vous faudra trouver une autre solution...

3.2.7 Charger des leçons

Le gros défaut de la version obtenu jusque là est qu'il faut modifier le source et recompiler pour ajouter des exercices dans l'interface. Ceci n'est évidement pas souhaitable. Pour changer cela, il faut bien entendu utiliser un mécanisme de plugin. La plus grosse partie du travail est déjà réalisée dans le template fourni. Vous trouverez deux leçons compilées sous forme de plugin : `logo.so` (qui reprend les exercices chargés en dur au lancement de l'interface) et `recursion.so` (qui contient des exercices auxquels vous n'avez pas encore accès). De plus, l'interface sait déjà changer de leçon dynamiquement et tout recalculer comme il se doit.

La seule chose qu'il vous reste à faire est de compléter la fonction `lesson_from_file()` dans le fichier `core/lesson.h` (une vingtaine de lignes devraient suffire). Le principal piège est qu'il ne faut pas fermer le module (avec `dlclose`) trop tôt. Dès que vous faites cela, toutes les données malloc()ées par des fonctions du module deviennent invalide. Il faut donc appeler `dlclose()` quand vous en avez fini avec cette leçon (ie, dans `lesson_free()`), pas avant. Vous allez sans doute devoir changer quelques petites choses de ci de là pour faire fonctionner ceci, mais rien de bien méchant.

Une fois ceci fait, vous pouvez charger dynamiquement des leçons. En particulier, vous avez accès aux exercices de récursion...

4 Extensions possibles

L'état du code à la fin de la section précédente constitue le minimum à atteindre pour avoir une note potable (entre 10 et 13 selon le degré de finition et le rapport). Pour aller au delà, il vous faudra implémenter certaines des extensions suivantes.

4.1 Persistance sur disque

Difficulté : très faible ; *Contexte :* il faut un peu plus de code, orienté système

Faites en sorte que le code des élèves soit sauvegardé avant sa compilation (quelque part sur le disque dur, par exemple dans `~/.clm/<leçon>/<exercice>`), et rechargé de façon transparente au lancement de l'exercice.

4.2 Multi-tortues

Difficulté : très faible ; *Contexte :* votre code système doit être plus générique

Les exercices proposés ne comportent qu'une tortue par monde, ce qui simplifie un peu l'implémentation. Écrivez un exercice comprenant plusieurs tortues, et vérifier que tout marche encore bien (y compris le bouton *Stop*).

4.3 Multi-mondes

Difficulté : faible ; *Contexte :* surtout des aspects systèmes, et un peu d'interface

Il s'agit d'autoriser les exercices à avoir plusieurs mondes parallèles. Il faut donc changer `w_init`, `w_curr` et `w_goal` en tableaux de monde. Un nouveau combobox dans l'interface permettra de passer de l'affichage d'un monde à l'autre. Vous pouvez vouloir ajouter des paramètres au monde, de façon à ce que le même code utilisateur se lance avec des paramètres différents selon le monde. JLM implémente cette fonctionnalité, et les exercices de récursion fournis seraient plus intéressant ainsi.

4.4 Exécution du code des élèves dans Valgrind

Difficulté : modérée ; *Contexte :* un peu de système, pas mal d'interface

Les erreurs les plus classiques en C se terminent souvent par un SEGFAULT. Il est assez difficile de comprendre puis corriger ces problèmes avec seulement `printf()` et Valgrind est souvent très utile pour cela. On souhaite donc donner la possibilité aux élèves utilisant CLE d'examiner leur code avec cet outil.

4.5 Vitesse d'animation configurable

Difficulté : faible ; *Contexte :* surtout de l'interface

Pour l'instant, une pause de 50ms est marquée chaque fois que la tortue ajoute un trait dans le monde. Il faudrait ajouter un `GtkHScale` ou équivalent pour permettre de régler cette valeur depuis l'interface.

4.6 Indiquer les erreurs de compilation avec des marqueurs graphiques

Difficulté : modérée ; *Contexte :* quasi uniquement de l'interface

Si l'on vise le confort d'utilisation offert par eclipse, il faut rapporter graphiquement les erreurs. Pour cela, on va ajouter des petits ronds rouges dans la marge de l'onglet présentant le source à toutes les lignes où des erreurs ont été rapportées par `gcc`, et faire en sorte que le message s'affiche quand on survole le petit rond rouge. Consultez la documentation de `GtkSourceView` en général, et celle de `GtkSourceMark` et `GtkSourceGutter` en particulier. Imiter le composant de droite des vues eclipse, qui montre toutes les erreurs dans le fichier même si elles ne sont pas à des numéros de lignes affichés, semble beaucoup plus dur (puisqu'il n'y a rien de tel dans Anjuta).

Si vous avez implémenté §4.4, remarquez qu'il est possible de demander à Valgrind d'utiliser un descripteur particulier pour ses affichage, voire de faire une sortie en XML. Cela peut permettre d'améliorer l'affichage de ces résultats.

4.7 Ajouter des tortues dynamiquement

Difficulté : importante ; *Contexte :* la mécanique système doit être retravaillée

Ajoutez une fonction que le code des élèves peut invoquer : `spawn(fonction)` qui permet de dupliquer la tortue et de lui faire exécuter le code de la fonction passée en paramètre. Une solution est de faire en sorte que les processus exécutant les tortues soient capables de fork()er eux aussi, et d'utiliser des tubes nommés (cf. `mkfifo(2)`) pour permettre au processus `CLE` de communiquer avec les nouveaux venus. Ce n'est pas forcément la seule approche, mais je pense que c'est la plus simple.

4.8 Templating d'exercice

Difficulté : faible ; un peu long ; *Contexte :* très utile aux auteurs d'exercice, sans rapport avec le système

La JLM permet de partager le code de l'entité solution (permettant de calculer le monde objectif) et le template du code présenté aux élèves. Cela marche en effaçant automatiquement des morceaux du code délimités par des marqueurs comme `/* TEMPLATE */`. Il serait bien pratique d'avoir le même genre de chose dans le CLE. N'hésitez pas à vous inspirer du comportement de la JLM en la matière.

4.9 Nouveaux types de monde

Difficulté : modérée à importante (et un peu long) ; *Contexte :* le modèle de données doit être retravaillé

Faites en sorte que le type `world_t` contienne des pointeurs vers des fonctions `free`, `copy`, `eq` et `turtle_add`, puis changez `world_free`, `world_copy`, `world_eq` et `world_turtle_add` pour simplement utiliser les fonctions pointées dans la structure `world_t`. Faites le même genre de modification à `turtle_t`.

Ensuite, créez un mécanisme de plugin permettant de changer le type de monde dynamiquement. Il faut donc que les exercices puissent changer le plugin de monde lors de leur chargement. Le monde des tris ou celui de Hanoi de la JLM sont suffisement simples pour que vous puissiez les adapter à notre contexte sans grande difficulté. Bien entendu, le monde des buggles est plus intéressant, même s'il est plus long à adapter. Le mieux étant sans doute d'imaginer des types d'exercices spécifiquement pensés pour le langage C...

4.10 Le reste...

Vous êtes libres d'ajouter toutes les idées d'extension que vous imaginez. Demandez moi avant de vous lancer si vous souhaitez avoir mon avis sur la complexité et l'intérêt d'une idée donnée. Si vous manquez d'imagination (mais pas de motivation), j'ai aussi quelques pistes à proposer aux plus courageux.