Université Paris Diderot
L3 Informatique

Conduite de Projet
Année 2019-2020

Burp [1] — Un jeu de programmation

version 1.1

The robots are controlled by a program [. . .]. The robot's mission is to seek out and destroy other robots, each running different programs.

— *https://en.wikipedia.org/wiki/Crobots*

1 Introduction

Le but de ce projet est d'implémenter en langage C un *jeu de programmation* [2]. Dans ce jeu, des robots s'entretuent dans une arène, chacun d'eux exécutant un script donné en début de partie. Le jeu se termine lorsqu'il ne reste plus qu'un seul robot encore fonctionnel dans l'arêne : son script est alors déclaré gagnant [3]. Le langage de script que nous utiliserons est une version très simplifiée de l'antique langage BASIC [4]. Votre programme devra permettre la simulation de l'arène, ainsi que l'interprétation en temps réel du script de chaque robot.

Les éléments de spécification qui suivent se divisent en un *projet minimal* (dont on s'attend à ce qu'il soit implémenté en intégralité) et des suggestions d'extensions, dont l'implémentation pourra consolider votre note.

Le projet est à faire en trinômes. La progression régulière de votre travail au cours du semestre, de même que le bon usage des outils et des pratiques de conduite de projet présentés dans ce cours, seront au moins aussi importantes pour votre évaluation que le résultat final. Les informations pratiques utiles vous seront fournies sur la page Moodle du cours. [5]

2 Projet minimal

A l'exception du nombre initial de robots, Les valeurs numériques que nous donnons dans cette partie (dimensions de l'arène, points de vie ou vitesse des robots, etc.) ne le sont qu'à titre indicatif : toutes ces valeurs devront être déclarées dans votre programme comme des constantes (`#define`) de valeurs librement modifiables avant sa compilation.

2.1 Arène de jeu

Le jeu se joue à 4 robots dans une espace virtuel, *l'arène*, de taille $10\,000 \times 10\,000$. Cet espace est supposé entouré de murs infranchissables.

Au début du jeu, chaque robot reçoit un script et est placé aléatoirement dans un des quatre coins de l'arêne. Les coordonnées des positions de robots sont réelles (`double`) et variables au

1. Burp – Basic Used for Robots Programming
2. https://en.wikipedia.org/wiki/Programming_game
3. Des versions très poussées de ce concept existent : CRobots, déjà cité ci-dessus, mais aussi RobotWar (https://en.wikipedia.org/wiki/RobotWar), ou encore Core Wars (https://corewars.org/).
4. https://en.wikipedia.org/wiki/BASIC
5. http://moodlesupd.script.univ-paris-diderot.fr/course/view.php?id=6660

cours du temps. A tout moment, un robot occupe un espace de taille 10×10 centré autour de sa position courante.

Pendant le déroulement du jeu, l'arêne pourra également contenir des missiles mobiles tirés par les robots, de coordonnées réelles et de taille ponctuelle.

2.2 Déroulement d'un cycle

A chaque cycle d'horloge : chaque robot encore fonctionnel exécute l'instruction courante de son script ; les positions des robots et des missiles pouvant être mises à jour sont mises à jour ; les missiles devant exploser explosent ; les robots subissent les dégats résultants d'explosions ou de collisions.

2.2.1 Exécution du script

On notera que si les coordonnées des robots et des missiles dans l'univers simulé sont réelles, le langage ne manipule que des nombres entiers (en particulier, le robot ne peut connaître que les approximations entières des coordonnées des autres robots, *c.f.* `GPSX`, `GPSY`). Le langage permet notamment à un robot d'effectuer l'une des deux actions suivantes.

Engine Démarrer son moteur en choisissant une vitesse et une direction donnée. La direction est spécifiée par un angle, la vitesse choisie comme un pourcentage (entre 0 et 100%). d'une certaine vitesse maximale égale à 10 unités par cycle.

Shoot Lancer un missile dans une direction donnée en spécifiant à quelle distance de la position de tir celui-ci devra exploser. La distance maximale d'explosion est de 7000 unités. Les missiles se déplacent à une vitesse de 500 unités par cycle.

2.2.2 Mise à jour des positions de robots

Chaque nouvau démarrage de moteur par un robot entraîne automatiquement le déplacement de celui-ci aux cycles suivants si la vitesse choisie est non nulle. La collision du robot avec les murs de l'arêne ou avec un autre robot entraîne son arrêt immédiat.

2.2.3 Mise à jours des positions de missiles

Les missile tirés sont ajoutés à l'arêne et sont automatiquement déplacés aux cycles suivants. Ils explosent et disparaissent de l'arêne soit immédiatement après avoir parcouru la distance prévue, soit en cas de collision avec un autre robot ou avec un mur.

Un robot ne peut avoir plus de 2 missiles en cours de déplacement : lorsque cette limite est atteinte, les nouvelles demandes de lancement sont ignorées, jusqu'à ce qu'au moins l'un de ces deux missiles explose.

2.2.4 Impact des collisions et explosions

Les robots commencent le jeu à 0% de dégats. Ils meurent et restent définitivement immobiles dans l'arêne lorsqu'ils atteignent ou dépassent 100% de dégats. La collision d'un robot avec un mur ou un autre robot entraîne 2% de dégats pour celui-ci (donc 2% pour chaque robot en cas de collision entre robots). L'explosion d'un missile entraîne :

— 3% de dégats pour chaque robot à moins de 400 unités du point d'explosion,
— 5% de dégats pour chaque robot à moins de 200 unités du point d'explosion,
— 10% de dégats pour chaque robot à moins de 50 unités du point d'explosion.

2.3 L'interpréteur

La syntaxe et la sémantique du langage de script des robots, le langage BURP, est décrit à l'annexe A page 7. Les extensions du langage décrites ci-dessous présupposent la lecture de cette grammaire.

2.4 GLib

La lecture et l'interprétation de la sémantique du langage de script des robots nécessitent du traitement de fichiers-textes, de la gestion de mémoire ainsi que l'usage de structures de données potentiellement complexes. Afin de de vous concentrer sur les aspects les plus intéressants du projet, nous vous recommandons de vous servir pour cette partie utilitaire d'une bibliothèque standard C étendue, comme par exemple GLib [2]. Préférez systématiquement l'usage de types et structures de données déjà implémentés dans GLib (ou l'équivalent) à l'implémentation manuelle des mêmes abstractions.

2.5 Interface visuelle

On s'attend à ce que le rendu visuel du programme soit en *mode texte* (pas d'interface graphique). Vous aurez donc besoin d'une bibliothèque externe permettant de réaliser une telle interface, pouvant s'afficher dans tout terminal. On vous demande ici d'utiliser la bibliothèque `ncurses`, largement utilisée par des projets libres. La présentation de cette bibliothèque est disponible sur la page du projet `ncurses` [3].

L'interface devra, pendant tout le déroulement du jeu, afficher en continu l'arène et son contenu, ainsi les statuts des 4 robots (position, dégats, statut des missiles en cours de tir, etc). Elle devra également permettre de : mettre le jeu en pause ; régler la vitesse du jeu (+/- ou encore couple de flèches, etc.) en affichant celle-ci en permanence dans la fenêtre.

2.6 Ligne de commande

L'invocation du programme se fera dans tous les cas en écrivant les scripts des robots dans des fichiers externes, par exemple, dans sa version minimale (*c.f.* les propositions d'extension ci-dessous) de la façon suivante :

```
./programname prog1.burp ... prog4.burp
```

les fichiers `prog1.burp` jusqu'à `prog4.burp` contenant les codes-sources des scripts des robots.

3 Extensions possibles

Afin de consolider votre note, voici une liste d'extensions possibles pour votre logiciel. Cette liste n'est pas exhaustive, et vous pouvez nous soumettre vos propres idées d'améliorations ou de nouvelles fonctionnalités. Les éléments que vous aurez traités devront être mentionnés dans votre rapport.

3.1 Extensions du langage

Le langage de script des robots est librement extensible, avec cependant une contrainte : vous devrez intégrer à votre projet les éléments permettant de traduire automatiquement (i.e., *compiler*) tout script écrit dans votre langage étendu vers le langage de base.

Plusieurs choix d'impléntation sont envisageables, mettant en jeu un ou plusieurs exécutables (partageant éventuellement du code commun mais compilé une seule fois) et une ou plusieurs phases de traduction :

— traducteur externe, prenant en argument le nom d'un code-source dans le langage étendu et produisant un code-source opératoirement équivalent écrit dans le langage de base ;

— traducteur intégré à l'interpréteur, l'interpréteur acceptant des codes-sources écrits dans le langage étendu, le traducteur travaillant sur leur représentation interne ;

— compilateur externe produisant une représentation du code-source sous forme de codes d'instructions et de données numériques (tokens), un traducteur externe produisant un nouveau code compilé exprimé dans les tokens du langage de base, l'interpréteur acceptant en ligne de commande les fichiers compilés produits par ce traducteur, etc.

Voici une liste d'idées d'extensions :

Commentaires

```
2 POKE 42 GPS X // CECI EST UN COMMENTAIRE
```

Numéros de lignes libres Les numéros des lignes deviennent libres, pourvu qu'ils forment une suite strictement croissante (`10`, `20`, `25`, etc.). La traduction effectue une renumérotation en `0`, `1`, `2`, etc.

Identifiants Plutôt que de manipuler des valeurs d'adresses explicites, on offre la possibilité de définir des identifiants pour ces valeurs.

letter := `A` | ... | `Z`
identifier := *letter{letter}*

Le programme commence par la déclaration de tous ses identifiants, avant sa ligne numéro `0`, par une suite d'instructions de la forme `PTR` *identifier address*, chacune déclarant un identifiant pour une valeur d'adresse. La traduction substitue chaque identifiant par son adresse associée—y compris, dans les expressions spécifiant des valeurs d'adresses dans les déclarations suivantes, *e.g.* :

```
PTR X 0
PTR Y 1
PTR I 2
PTR TABX 3
PTR TABY (TABX + CARDINAL)
PTR A (TABY + CARDINAL)
0 ENGINE RAND 360 RAND 100
1 POKE I 0
2 POKE A 42
....
```

Notations abrégées On ajoute les raccourcis d'incrémentation et de décrémentation à la C, avec leur sens habituel, un opérateur d'affectation, etc.

- *address*`++`
- *address*`--`
- `*`*address* pour `PEEK` *address*
- *address* `:=` *value* pour `POKE` *address value*.
- etc.

Connecteurs booléens

- `(`*condition* `AND` *condition*`)`
- `(`*condition* `OR` *condition*`)`
- `NOT` *condition*

Commandes multiples

```
2 IF X > 0 THEN 17; IF X = 0 THEN 20; GOTO 24
```

Itérations

- num_1 `FOR` *address* `=` *first* `TO` *last*

  ⋮

  num_2 `NEXT`

L'instruction `NEXT` incrémente la valeur écrite à *address*, puis, si elle n'est pas à sa valeur maximale, va à la ligne suivant num_1. Ajout éventuel d'un `STEP`.

- num_1 `WHILE` *condition*

  ⋮

  num_2 `DONE`

4 Évaluation de votre projet

La notation prendra en compte tous les points suivants.

Utilisation des outils standard Les cours magistraux présenteront tout au long du semestre des outils standard de développement (Makefile, git, etc.) qui devront être *effectivement* utilisés pendant le projet. Cette utilisation va de pair avec une séparation des différentes parties du projet en modules et librairies séparées.

Modularité et tests Les différents modules du programme devront être séparés en différents fichiers sources (`.c`) et en fichiers d'en-tête (`.h`) utilisés à bon escient. Vous préparerez aussi des tests unitaires de chaque fonction importante sur des exemples choisis, et des tests d'acceptation des fonctionnalités globales du projet. Ces tests seront inclus dans votre dépôt git.

Contrôle continu Le travail doit être réalisé tout au long du semestre grâce à un répertoire *git* pour chaque trinôme.

1. Votre répertoire doit être *privé*, mais dès les premières semaines, vous donnerez accès à ce répertoire à tous les enseignants. Ceci permettra notamment de garder la trace et les dates de contributions de chaque étudiant.

2. Vous pouvez choisir entre le service d'hébergement GitLab offert par l'université [6] ou tout autre service tiers, toujours en respectant les consignes du point précédent.
3. Nous vous demandons de gérer en continu les modifications de votre répertoire et d'effectuer systématiquement des revues de code à l'intérieur de votre trinôme :
   — avec GitLab, à l'aide de *merge requests* [7]
   — avec GitHub, à l'aide de *pull requests* [8]

Diagramme d'architecture Vous devrez réaliser, dès que possible, un diagramme décrivant l'architecture de votre projet—au moins le graphe de dépendance de ses modules, étiqueté par leurs noms et une description synthétique de leurs fonctionnalités respectives. Ce diagramme servira de base à votre soutenance.

Journal Chaque trinôme tiendra à jour un *journal de développement* inclus dans son répertoire git. Chaque semaine, vous indiquerez dans votre journal vos objectifs pour la semaine à venir, et vos résultats de la semaine passée. Dans la mesure du possible, faites en sorte que chaque commit corresponde à une tâche précise et soit effectué par la personne ayant effectué la tâche. Un dépôt git n'est pas limité à du code, n'hésitez pas à y inclure d'autres documents qui nous permettront de suivre votre travail, par exemple : des courts résumés de prise de décision, des notes sur le fonctionnement des bibliothèques que vous allez utiliser, la structure que vous projetez pour vos modules, etc.

Lisibilité et clarté Il est fortement recommandé d'indenter systématiquement, d'éviter les lignes trop longues (80 caractères), de choisir des noms parlants pour les variables et fonctions, et de commenter le code souvent, mais en restant concis. Vous supprimerez également toute duplication de code, suivant le principe DRY [1] ("Don't Repeat yourself").

Références

[1] Don't Repeat Yourself. https://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

[2] Glib reference manual. https://developer.gnome.org/glib/stable/.

[3] Ncurses (page officielle). http://www.gnu.org/software/ncurses/ncurses.html.

6. http://moule.informatique.univ-paris-diderot.fr:8080/
7. https://docs.gitlab.com/ee/gitlab-basics/add-merge-request.html
8. https://help.github.com/articles/about-pull-requests/

A Le langage BURP

A.1 Grammaire du langage

Chaque définition (*program*, *command*, *line*, etc.) décrit une certaine catégorie de suite de symboles, éventuellement de plusieurs formes séparées par |. Les symboles littéraux sont notés `A`, `0`, `+`, etc. La notation *{catégorie}* signifie : une succession quelconque d'éléments de *catégorie*.

digit := `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
integer := [`-`] *digit{digit}*
number := *digit{digit}*
operator := `+` | `-` | `*` | `/` | `%`
expression :=
  integer
| `(`*expression operator expression*`)`
| `PEEK` *expression*
| `RAND` *expression*
| `CARDINAL`
| `SELF`
| `SPEED`
| `STATE` *expression*
| `GPSX` *expression*
| `GPSY` *expression*
| `ANGLE` *expression expression expression expression*
| `TARGETX` *expression expression expression*
| `TARGETY` *expression expression expression*
| `DISTANCE` *expression expression expression expression*
comparison := `<` | `<=` | `=` | `<>` | `>=` | `>`
condition := *expression comparison expression*
command :=
  `WAIT` *expression*
| `POKE` *expression expression*
| `GOTO` *number*
| `IF` *condition* `THEN` *number*
| `ENGINE` *expression expression*
| `SHOOT` *expression expression*
line := number command
program := line{ line}

Les opérateurs binaires ont leur sens habituel (`%` désigne le modulo), et ne s'emploient dans une expression qu'avec leur parenthésage complet. Les numéros de lignes successifs d'un programme doivent former la suite des entiers `0`, `1`, `2`, etc.

A.2 Sémantique

Le sens des instructions est ici décrit en précisant la nature de leurs arguments numériques (*size*, *angle*, *speed*, coordonnées x, y etc.) : il est clair que tous ces noms désignent la catégorie *expression*, *i.e.* peuvent être remplacés par des expressions quelconques.

A.2.1 Attente

— `WAIT` *delay*. Le robot exécutant attend pendant *delay* cycles d'horloge avant de passer à la ligne suivante de son script. Il poursuit son déplacement si sa vitesse est non nulle, de même que tous ses missiles lancés (*c.f.* ci-dessous).

A.2.2 Accès à la mémoire

Chaque robot dispose d'une unique zone mémoire suffisamment grande, dont les adresses sont numérotées à partir de 0. Cette zone-mémoire est initialisée à des valeurs nulles.

— `POKE` *address value* écrit *value* à l'adresse *address*.

— `PEEK` *address* vaut la valeur lisible à l'adresse *address*.

Les accès par `POKE` et `PEEK` sont vérifiés, avec une erreur d'exécution en cas d'accès fautif (adresse invalide).

A.2.3 Sauts conditionnel et inconditionnel

— `GOTO` *num*. Poursuivre l'exécution à l'instruction numéro *num*.

— `IF` *condition* `THEN` *num*. Si la condition est vérifiée, poursuivre l'exécution à l'instruction numéro *num*.

Le numéro doit être dans les bornes du script.

A.2.4 Tirage aléatoire

— `RAND` *num* vaut un nombre aléatoire tiré dans l'intervalle $[0, \ldots, num[$.

Le nombre doit être strictement positif.

A.2.5 Contrôle et examen de l'état des robots

— `CARDINAL` vaut le nombre de robots initialement présents dans l'arène.

— `SELF`. Tous les robots portent des numéros distincts, de `0` à `CARDINAL - 1`. Cette expression vaut le numéro du robot exécutant.

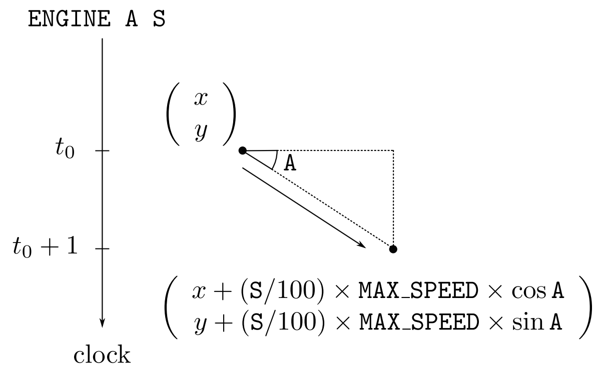
— `SPEED` vaut la vitesse courante du robot exécutant – celle-ci peut devenir nulle si le robot heurte un obstacle (un autre robot, ou l'un des murs).

— `STATE` *num* vaut le nombre de points de vie du robot n°*num*.

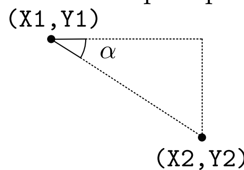
— `GPSX` *num*, `GPSY` *num* valent respectivement l'abscisse, et l'ordonnée du coin en haut à gauche du robot n°*num*. Ces coordonnées sont des entiers.

— `ENGINE` *angle speed*. Permet d'orienter le robot selon un angle donné relativement à l'axe des abscisses, en réglant son moteur sur un certain pourcentage (entre 0 et 100%) de sa vitesse maximale.

A chaque tour de jeu, le robot se déplacera de la manière décrite dans le diagramme ci-dessous – jusqu'à ce qu'il rencontre un obstacle, où encore change son orientation ou sa vitesse.

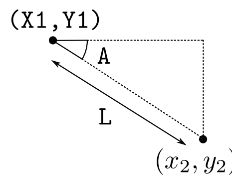


- `SHOOT` *angle distance* permet au robot exécutant de tirer un missile suivant un angle entier donné. Le missile explosera dans tous les cas après avoir parcouru la distance spécifiée, ou avant cette distance s'il heurte un autre robot ou un mur.
  Les dégats d'une explosion sont répercutés sur chaque robot à proximité, et sont d'autant plus importants que cette explosion est proche.
- `ANGLE` x_1 y_1 x_2 y_2 vaut l'angle formé par le vecteur unitaire $(1, 0)$ et le vecteur allant de (x_1, y_1) à (x_2, y_2), converti au degré entier le plus proche.



$\alpha =$ `ANGLE X1 Y1 X2 Y2`

- `TARGETX` x_1 *angle length* et `TARGETX` y_1 *angle length* valent respectivement l'abscisse et l'ordonée (toujours converties en les entiers les plus proches) du point (x_2, y_2) tel que le vecteur allant de (x_1, y_1) à (x_2, y_2) soit de longueur *length* et forme un angle de *angle* avec le vecteur unitaire $(1, 0)$.



$x_2 =$ `TARGETX X1 A L`
$y_2 =$ `TARGETY Y1 A L`

- `DISTANCE` x_1 y_1 x_2 y_2 vaut la distance entre les points (x_1, y_1) et (x_2, y_2), convertie en l'entier le plus proche.

Les numéros de robots doivent être valides. Les coordonnées doivent être dans les bornes de l'arêne. Un pourcentage de vitesse doit être compris entre 0 et 100.