

L'objectif de ce projet est de créer un interpréteur pour un langage permettant de déplacer une ou plusieurs tortues dans un jardin. Les spécificités du langage sont détaillées dans la suite du sujet.

Le projet peut se faire seul ou en binôme et est à rendre au plus tard le **vendredi 23 décembre** sur le dépôt "Projet 2022-2023 turtl" sur moodle.

Généralités

Le projet est divisé en deux parties : la reconnaissance du langage et l'interprétation du langage.

La reconnaissance du langage sera effectuée dans un premier temps. Les explications nécessaires à l'interprétation du langage (par exemple l'interface graphique) seront fournis le lundi 7 novembre lors d'une mise à jour du sujet.

Pour réaliser votre projet vous pouvez récupérer l'archive **tortue.zip** sur moodle contenant la base du projet. Un dossier exemple vous sera proposé le 7 novembre contenant des fichiers devant être interprétés et le résultat associé.

Prérequis

Afin de pouvoir compiler le projet il est nécessaire d'avoir installer les paquets suivant (sur Linux) :

- flex
- libfl-dev
- bison
- qt5-qmake
- qtbase5-dev (ou qt5-default pour les versions plus anciennes de Linux)

Pour Mac OS X les paquets doivent être similaires.

Pour compiler nous utilisons qmake (équivalent à CMake) car l'interface graphique est programmée en QT et nécessite une compilation spéciale, il est conseillé de le faire à partir du terminal en créant un répertoire build à l'intérieur du projet puis de faire :

- qmake ../projet.pro
- make
- ./tortue

Lors de l'exécution une fenêtre s'ouvre, pour cette première partie il n'est pas demandé d'interagir avec cette fenêtre, vous pouvez traiter l'ensemble du langage dans le terminal comme dans les premiers TP.

Pour ajouter des fichiers à votre projet il faut les ajouter au fichier tortue.pro pour les fichiers concernant votre structure de données et ajouter vos fichiers à expressions/expressions.pro pour les fichiers liés à la construction de l'arbre syntaxique.

Il est possible d'utiliser QtCreator pour compiler mais il est possible que celui-ci ne détecte pas le fichier de bibliothèque libexpressions.a créé par le fichier expressions/expressions.pro. Dans ce cas vous devez effectuer une première compilation dans un terminal pour que le fichier libexpressions.a soit détecté par QtCreator. Attention donc si vous modifiez le dossier expressions à bien recompiler entièrement votre projet dans un terminal.

Langage

Instructions simples

La tortue peut être déplacée en utilisant des instructions (Exemple 1).

Il n'est possible d'utiliser qu'une seule instruction par ligne. Une tortue peut :

- avancer d'une case avec l'instruction `avance [nombre]`
- reculer d'une case avec l'instruction `recule [nombre]`
- sauter (avancer de deux cases en sautant par dessus une case occupée) avec l'instruction `saute [nombre]`
- tourner d'un quart de cercle avec l'instruction `tourne <à droite|à gauche>`¹

Les instructions ont un argument facultatif correspondant au nombre de fois à effectuer une instruction. Ce nombre peut être exprimé par une constante entière ou une expression arithmétique utilisant les opérateurs suivant +, -, *, / et (). Si le résultat n'est pas un entier on arrondira à l'entier le plus proche. Le nombre de répétitions est placé après l'instruction et peut être suivi du mot `fois`.

Ainsi, les instructions suivantes permettent toutes d'avancer la tortue de 2 cases :

- `avance 2 fois`
- `avance 2`
- `avance (3*3+1)/5 fois`

Lorsqu'il existe plus d'une tortue, les instructions s'appliquent à toutes les tortues. Il est cependant possible d'affecter des actions à une tortue donnée en ajoutant `@n` à la fin d'une instruction, où `n` correspond au numéro de la tortue ciblée (Exemple 7 et Exemple 8).

Conditionnelles et boucles

Il est possible de spécifier une série de déplacements sous certaines conditions (Exemple 2). Les deux conditions possibles sont des fonctions permettant de déterminer si une case adjacente donnée est vide ou si elle contient un mur. Elles s'écrivent respectivement `mur <direction>` et `vide <direction>`. Il existe quatre directions possibles : `devant`, `derriere`, `à droite`, `à gauche`.

La négation peut être utilisée devant une condition, avec le mot clé `pas de`. Par exemple, la condition `pas de mur devant` utilisée dans une conditionnelle permet de tester s'il y a un mur devant la tortue.

Une conditionnelle s'écrit de la manière suivante : `si <condition>: ... fin si`

Il est possible d'ajouter un bloc `sinon` : pour spécifier les instructions à effectuer si la condition testée est fausse.

Deux types de boucles sont utilisables. La première permet d'effectuer une série d'instructions tant que la condition testée est vraie et s'utilise de la manière suivante :

`tant que <condition>: ... fin tant que`

La seconde permet de répéter `n` fois une série d'instructions. Comme précédemment, `n` est un entier positif et peut correspondre à une expression arithmétique. Elle s'utilise de la manière suivante :

`repete <nombre >: ... fin repete`

Fonctions

La définition de fonctions en dehors du programme principal est possible (Exemple 4). Une fonction se déclare comme suit : `fonction <nom>: ... fin fonction`

Si la fonction prend des arguments en paramètre, l'accès au premier argument dans la fonction se fait avec `$1` et de manière générale, l'argument `n` se fait avec `$n`.

Il n'existe pas de limite au nombre d'arguments d'une fonction. Il n'est pas possible de créer deux fonctions avec le même nom. Une fonction est appelée avec son nom et d'éventuels arguments. Par exemple, `test 1 2 3` permet d'appeler une fonction `test` avec les arguments 1, 2 et 3.

La fonction `main` est le point d'entrée du programme et se déclare comme les autres fonctions.

Instructions spéciales

Dans le programme principal, il est possible d'utiliser certaines instructions particulières qui modifient les

paramètres de l'affichage. Par exemple on peut changer la couleur de la tortue dans le programme principal avec le mot clé couleur (Exemple 6). Il existe deux zones de couleurs la carapace et le motif, si aucune zone n'est précisée la couleur s'applique à la carapace.

Les couleurs sont données en hexadécimal au format #rrggbb. Par exemple, couleur carapace #aBcDeF. On peut spécifier une couleur pour une tortue donnée en ajoutant @n à la fin (Exemple 7). Un nombre² de tortues peut être défini par tortues <nombre>. Cette instruction est facultative et peut être utilisée au plus une fois. Si elle n'est pas utilisée, il y aura une tortue par défaut.

Le jardin sur lequel déplacer les tortues peut être choisi avec l'instruction jardin '<fichier>' (Exemple 7). Ce fichier permet de définir la taille du jardin ainsi que les positions des murs.

Enfin, il est possible d'utiliser des commentaires en les préfixant avec --.

Interprétation du langage

Afin que vous vous concentriez sur la reconnaissance du langage nous omettons cette section pour l'instant. Une seconde version du sujet avec cette partie vous sera communiquée. Cette nouvelle version vous présentera les fonctions permettant d'interagir avec l'interface graphique. Vous avez déjà accès à tous les fichiers permettant de faire fonctionner l'interface graphique dans le dossier GUI, il ne sera jamais nécessaire de modifier ces fichiers. Vous pouvez tout de même consulter leur contenu.

Pour cette première partie vous pouvez construire entièrement votre grammaire et la structure de données associée.

Notation

Votre projet doit se constituer ainsi :

- Analyseur lexical (scanner.ll)
- Analyseur syntaxique (parser.yy)
- Structure de données (fichiers à créer et utilisation des expressions)
- Interaction avec l'interface graphique

Il est important d'avoir une structure de données intermédiaire afin de pouvoir avoir une entrée et une sortie différente. Vous devez donc modéliser votre jardin et vos tortues dans cette structure intermédiaire.

Le projet sera noté principalement sur votre parser et votre structure de données. La partie analyse lexicale (élaboration des expressions régulières dans le fichier scanner.ll) ne compte que pour une petite partie de la note finale. La sortie de votre parser (interaction avec l'interface graphique) sera comptée uniquement en points bonus. La majorité des points est basée sur votre analyseur syntaxique (fichier parser.yy) et votre structure de donnée intermédiaire.

Un projet ne contenant que l'analyseur syntaxique et l'analyseur lexical ne sera pas bien noté il faut ABSOLUMENT avoir une structure de données intermédiaire.

Exemples

```
1 fonction main :
2 -- ceci est un commentaire
3 -- il y a 3 actions de base : avance , recule et saute
4 -- ces actions ont un argument facultatif qui est le nombre
5 -- de fois où l'action doit être répétée
6
7 avance 1 fois -- avance 1 fois
8 recule -- recule 1 fois
9 saute 2 fois
10 recule 2 -- " fois " est facultatif
11 f in fonction
```

Exemple 1: Exemple de programme avec des actions de déplacement base.

```
1 fonction main :
2 avance 2 fois
3
4 s i mur devant : -- une condition
5 -- il y a deux fonctions utilisables dans les conditions
6 -- mur <direction > et vide <direction >
7 -- avec <direction > = [à gauche |à droite | devant | derriere ]
8
9 tourne à gauche
10 -- la tortue tourne d'un quart de cercle à gauche ou à droite
11
12 -- tourne <à gauche |à droite > [ nombre de fois ]
13
14 avance
15 s i mur derriere :
16 recule
17 saute
18 sinon : -- il est possible de donner un bloc 'sinon ' aux 'si '
19 saute
20 avance
21 f in s i
22 f in s i
23
24 -- comme avance / recule / saute tourne prends un param è tre facultatif
25 tourne à gauche 2 fois
26
27 avance 3
28 f in fonction
```

Exemple 2: Exemple de programme avec des conditions.

```

1 fonction main :
2 repete 5 fois :
3 s i vide devant :
4 avance
5 sinon :
6 tourne à gauche
7 f in s i
8 f in repete
9
10 recule 2
11 f in fonction

```

Exemple 3: Exemple de programme avec des répètes.

```

1 fonction main :
2 avance 4 fois
3
4 tant que pas de mur devant :
5 -- les conditions peuvent être niées avec " pas de"
6 avance
7 f in tant que
8
9 -- appel de fonction
10 foo 1 2 3
11 -- pas de limite au nombre d'argument
12 -- /\ contrairement aux fonctions prédéfinies
13 -- /\ 'foo ' n'est pas équivalent à 'foo 1'
14 f in fonction
15
16 fonction foo : -- il est possible de créer des fonctions
17 avance $1 fois
18 -- $1 , $2 , .., $n , font références aux arguments de la fonction
19 recule $2
20 saute $3 fois
21 f in fonction

```

Exemple 4: Exemple de programme avec des fonctions.

```

1 fonction main :
2 avance 5*(1+2) fois
3 recule 1/(2□3)
4 bar 1 2+2 4
5 f in fonction
6
7 -- on peut utiliser des expressions à base de +,-,*,/ et ()
8 -- dans les arguments des actions / fonctions
9
10
11 fonction bar :
12 saute $1 * ( $2 + $3 )
13 f in fonction

```

Exemple 5: Exemple de programme avec des expressions arithmétiques.

```

1 fonction main :
2 -- on peut changer la couleur de la tortue
3 -- il y a deux couleurs ( carapace et motif )
4
5 -- les couleurs sont donn ées en hexad é cimal au format # RRGGBB
6 -- comme en CSS
7
8 couleur carapace #aBcDeF
9 couleur motif #282828
10
11
12 couleur #424242 -- é quivalent à 'couleur carapace #424242 '
13
14
15 f in fonction

```

Exemple 6: Exemple de programme avec des couleurs.

```

1 -- on peut gérer plusieurs tortues dans un même script
2
3 fonction main :
4 tortues 2 -- on définit le nombre de tortues
5 jardin '../jardin01.jdn'
6 couleur #123456 @1 -- @n à la fin des instructions permet
7 avance 1 fois @1 -- de choisir la tortue affectée
8 tourne à droite @2
9
10 avance -- toutes les tortues sont affectées si rien n'est précisé
11 fin fonction

```

Exemple 7: Gestion de plusieurs tortues 1.

```

1 -- certaines actions ne peuvent pas avoir d'indications de tortue
2
3 fonction main :
4 tortues 2
5
6 si mur devant @2 :
7 -- on peut ajouter l'indication de tortues dans les conditions
8 avance
9 recule @1
10 fin si
11
12 si pas de vide devant @1 :
13 avance
14 fin si
15
16 si mur à gauche :
17 -- si on ne précise pas de tortue, la condition équivaut à
18 -- (mur gauche @1) et (mur gauche @2) et .. et (mur gauche @n)
19 avance
20 fin si
21
22
23 -- tortues 2 @2 -- interdit
24
25 -- si mur devant : @1 -- interdit
26 -- avance 2
27 -- fin si
28
29 -- repete 4 fois : @1 -- interdit
30 -- recule
31 -- fin repete
32
33 -- tant que vide derrière : @3 -- interdit
34 -- recule
35 -- fin tant que
36
37 fin fonction

```

Exemple 8: Gestion de plusieurs tortues 2.

```

1 fonction main :
2 avance fois -- il faut préciser un nombre avant 'fois '
3
4 recule avance -- une seule instruction par ligne
5
6 recule 4+ -- expression arithmétique non correcte
7
8 couleur #1010EG -- pas au format hexadécimal
9 couleur 3 fois
10
11 tortues 5 fois -- pas de 'fois ' sur le nombre de tortues
12
13 avance 4 fois @4 fois -- pas de 'fois ' après le @ tortue
14
15 s i 4 : -- 4 n'est pas une condition valide
16 avance 3
17 f i n s i
18
19 avance 4 fois + 2 fois -- ça c'est pas valide
20
21 mur devant -- une condition n'a rien à faire ici
22
23 avance 0xFF -- que des nombres en base 10
24
25 s i mur devant : -- pas de 'fin si '
26 avance
27
28 fonction foo : -- une fonction n'a rien à foutre ici
29 avance
30 f i n fonction
31
32 f i n fonction

```

Exemple 9: Plein de trucs à ne pas faire.

1. Dans le sujet on suivra le formalisme bash, les paramètres obligatoires sont notés entre < et > et les paramètres facultatifs entre [et]. ↩️
2. Peut être une expression arithmétique ↩️