UNIVERSITÉ CATHOLIQUE DE LOUVAIN
FACULTÉ DES SCIENCES APPLIQUÉES
DÉPARTEMENT D'INGÉNIERIE INFORMATIQUE

La programmation en première année basée sur l'enrichissement progressif de micromondes multi-agents

Mémoire présenté en vue
de l'obtention du grade
de Licencié en Informatique
par Isabelle Cambron
et Mathieu Cuvelier

Promoteur : Peter Van Roy

Louvain-la-Neuve
18 mai 2006

Nous tenons à remercier tout particulièrement les personnes suivantes, qui nous ont aidées dans la réalisation de ce mémoire :

Notre promoteur, Peter Van Roy, pour nous avoir suivi tout au long de l'année.
Nos lecteurs, Kims Mens et Manuel Kolp.
Les assistants-chercheurs de l'équipe de Peter Van Roy et en particulier Yves Jaradin pour nous avoir aidés à résoudre de nomnbreux problèmes d'implémentation.
Le secrétariat INGI ainsi que Chantal Poncin grâce à qui nous avons pu donner les séances de cours de notre mémoire dans les meilleures conditions.
Thomas Vanstals qui nous a fait gagner un temps précieux lors de notre rédaction, en nous faisant part de son expérience avec LaTeX.
Les étudiants volontaires qui ont participé activement aux séances de cours et qui nous ont permis d'obtenir un premier feedback sur la méthode.
Véronique Heureux ainsi que Suzanne et Emmanuel Cambron pour avoir eu le courage de relire et corriger notre mémoire.
Et pour finir nos parents qui nous ont soutenus tout au long de nos études.
Merci !

Table des matières

Introduction

L'enseignement de l'informatique et de la programmation est le challenge du siècle à venir. La pédagogie de l'informatique en est encore à ses balbutiements, et en est appelée à évoluer tout le temps en fonction des nouvelles technologies. Mais parfois il faut s'arrêter et penser à changer complètement de perspective : plutôt que d'améliorer les méthodes d'enseignement existantes, une approche novatrice peut être tentée.

Le sujet qui nous occupe, l'apprentissage de la programmation par des étudiants de première année, sera abordé comme un apprentissage de micromondes autonomes successifs. La changement de perspective consistant en ne plus apprendre la programmation par le biais d'un langage unique, mais plutôt d'en apprendre les concepts clés avant de se lancer dans l'apprentissage d'un langage en particulier.

Philosophie de notre approche

Dans la cadre de ce mémoire visant à présenter une méthode pédagogique permettant de réaliser une approche conceptuelle dès la première année à l'université, nous avons utilisé l'approche des micromondes. Cette approche encore peu utilisée dans l'enseignement d'aujourd'hui a pour avantage de permettre de limiter les concepts enseignés dans chaque monde, sans déborder les étudiants. Le principe est d'introduire un micromonde, et donc d'introduire l'ensemble des concepts que contient ce micromonde. Une fois que les limites du micromondes sont atteintes, c'est-à-dire une fois que l'ensemble des concepts a été exploré, alors seulement il est possible de passer au prochain micromonde qui contient un ou des nouveaux concepts. Les limites sont ainsi sans cesse repoussées.

La concurrence va être un concept central de notre approche. En effet, comme un micromonde est un monde composé d'agents, il est important que la concurrence soit introduite très tôt. Il faut pouvoir expliquer ce qui se passe quand différents agents interagissent entre eux.

Le séquencement des concepts abordés ainsi que le micromonde final ont une grande importance. Dans ce dernier micromonde seront, en effet, rassemblés les connaissances accumulées par les étudiants tout au long des micromondes successifs. L'objectif de nos micromondes sera que les étudiants puissent programmer dans ce micromonde final de nouveaux agents qu'ils comprennent et qu'ils puissent utiliser la programmation multi-agents.

L'objectif final de l'ensemble de ce cours (qui sera atteint en principe en juin 2007) sera

que les étudiants puissent programmer dans un micromonde des composants concurrents, des composants eux-mêmes composés d'autres composants et qu'ils puissent traiter les pannes de manière efficace.

Contexte

Le monde est fait d'agents. Il suffit de regarder autour de soi pour s'en rendre compte. Les agents sont partout. Etudiants, chats, chiens, marchands, arbres, avions, molécules, baleines... Le monde est un ensemble d'agents. Certains agents sont composés de plusieurs agents plus petits, et ce à l'infini. Les agents peuvent également se rassembler pour créer de plus grands agents. Nous vivons dans un monde orienté agents.

Contrairement aux approches plutôt orientées objets qui sont enseignées actuellement, les agents peuvent interagir entre eux. Cette interaction introduit un nouveau concept extrêmement important en programmation : la concurrence. La concurrence apparaît lorsque deux ou plusieurs activités indépendantes peuvent être exécutées simultanément. Il ne devrait y avoir aucune interférence entre ces activités, à moins que le programmeur ne décide qu'il est nécessaire qu'elles communiquent entre elles.

La programmation orientée agents est aujourd'hui malheureusement encore trop peu utilisée. Non pas parce que cette approche est considérée comme moins bonne que d'autres, mais surtout parce que l'orienté agent est encore trop peu connu. De ce fait, les outils nécessaires pour la concurrence -fort utilisée et très importante dans le contexte des agents- sont encore trop complexes pour des débutants et trop peu évolués. Peu de moyens sont actuellement mis en oeuvre pour promouvoir cette nouvelle façon de programmer. Mais avec l'évolution des techniques de programmation, de plus en plus de programmes vont être des programmes d'agents qui se parlent entre eux. Les agents, et par conséquent la concurrence, se retrouveront donc partout.

Description du travail

Les cours actuels de programmation en première année contiennent souvent beaucoup de syntaxe assez complexe, des "recettes toutes faites" que les étudiants ne comprennent pas mais doivent appliquer. L'enseignement de la programmation est souvent limitée à la programmation orientée objet et contient peu de fond scientifique. Depuis plusieurs années, une autre approche est développée au sein du département INGI, basée sur les concepts des langages de programmation. Cette approche montre la programmation comme une discipline unifiée avec un fond scientifique. Mais cette approche n'a été développée que pour les étudiants à partir de la seconde année. Nous nous sommes donc fixé comme objectif d'étendre cette approche aux étudiants de première année également.

Objectifs

Le but de ce mémoire sera donc d'étendre l'approche 'conceptuelle' aux cours de première année. La première année est une année très différente des années suivantes car

les étudiants viennent d'horizons très variés, et ont chacun des connaissances différentes des autres. Il faut donc trouver une approche adaptée aux étudiants qui font leurs premiers pas dans le monde de la programmation, et trouver comment ils peuvent être aidés par une organisation basée sur les concepts des langages de programmation.

Un premier objectif sera de développer un cours adapté à ces étudiants qui arrivent en première année de baccalauréat à l'université. Ce cours exposera d'une façon aussi claire et simple que possible les concepts de base de la programmation.

Un second objectif sera de créer un monde conceptuel plus simple que celui qui existe actuellement. Le cours sera donc accompagné d'un logiciel interactif et vivant qui permettra de faire des exercices amusants en rapport avec les concepts enseignés. Ce logiciel utilisera plusieurs agents qui communiquent entre eux.

Structure du document

Nous allons commencer par exposer différentes **approches actuelles** utilisées dans l'enseignement de la programmation aux étudiants de première année en relevant quelques problèmes qui font que ces approches ne sont pas toujours efficaces. Nous décrirons ensuite plus en détail trois approches qui sont utilisées pour enseigner la programmation ainsi que leurs points forts et points faibles : l'approche du langage générique, l'approche fonctionnelle et l'approche micromondes.

Nous poursuirons en exposant **l'approche par enrichissement de micromondes** que nous avons développée. Cette section reprend l'ensemble des cinq micromondes développés en détails, suivis d'un fil conducteur pour deux micromondes qui pourraient être créés dans le futur.

Nous aborderons ensuite plus en détails l'**environnement de programmation LogOz**. Nous parlons de l'architecture du logiciel, justifierons certains choix et expliquerons certains aspects plus en détails, y compris ceux qui ne sont pas toujours visibles au premier abord.

Suivront les différentes **séances de cours** que nous avons développées avec des explications précises pour chacune d'elles, un aperçu des cours qui doivent encore faire l'objet d'une recherche, ainsi qu'une **évaluation** des séances de cours que nous avons données à quelques étudiants volontaires. Grâce à leur feedback, nous avons pu réaliser certains changements et améliorations aux cours ainsi qu'au logiciel que nous avons développé.

Pour terminer, nous présenterons les différentes **conclusions** auxquelles nous avons abouti et ce qu'il reste encore à faire.

En annexe, vous trouverez l'ensemble des cours donnés aux étudiants, le code du prototype ainsi qu'un manuel pour l'utiliser.

Chapitre 1

Approches actuelles

Ce chapitre a pour but d'expliquer ce qui nous a poussés à développer une nouvelle approche d'enseignement utilisant les micromondes. Nous allons tout d'abord commencer par exposer le problème des méthodes d'enseignement actuelles de la programmation en tentant de comprendre ce qui ne va pas et ce qui pourrait être amélioré.

Nous allons ensuite exposer trois méthodes différentes qui sont utilisées aujourd'hui pour enseigner la programmation à des étudiants débutants : l'approche orientée objets, l'approche fonctionnelle et l'approche basée sur des micromondes. Chacune de ces approches possède ses points forts et ses points faibles, aussi bien d'un point de vue pédagogique que d'un point de vue informatique.

1.1 Les problèmes

L'approche actuelle utilisée dans l'enseignement de la programmation pour les étudiants de première année comporte différents grands problèmes. Tout d'abord, les étudiants n'ont pas tous le même niveau en arrivant à l'université. Certains n'ont encore jamais touché à la programmation alors que d'autres ont déjà certaines connaissances, plus ou moins avancées. Il faut donc mettre sur pied un moyen d'apprendre les bases aux étudiants les moins avancés, tout en essayant d'intéresser un maximum les étudiants ayant déjà vu ces bases auparavant. En plus de cet enseignement de base, il faut leur fournir les moyens qui permettront à tous de suivre par la suite des cours plus évolués de programmation et de structures de données.

Un autre problème est le choix de l'approche utilisée par l'enseignant. Nous développerons dans le point qui suit différentes approches possibles qui sont utilisées actuellement. Aujourd'hui c'est l'industrie qui pousse les universités à enseigner les langages qu'elle utilise (tels que le JAVA ou le C++) afin que les étudiants soient rentables dès qu'ils sont engagés à la fin de leurs études. Mais cette façon d'agir empêche l'université de faire de la recherche sur de nouvelles techniques d'enseignement et de programmation. L'université devrait aussi pouvoir inventer de nouveaux concepts qui seront par la suite utilisés dans les entreprises. Les recherches des deux côtés en parallèle pourraient aboutir à une saine

concurrence fort bénéfique pour le monde de l'informatique.

1.2 Approches utilisées

1.2.1 Approche orientée objets

La programmation orientée objets [1] consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel, que l'on appelle domaine, en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objets. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, ...).

L'approche objet est une idée qui a désormais fait ses preuves. Simula a été le premier langage de programmation à implémenter le concept de classes en 1967. En 1976, Smalltalk implémente les concepts d'encapsulation, d'agrégation, et d'héritage qui sont les principales caractéristiques de l'approche objet. D'autre part, de nombreux langages orientés objets ont été mis au point dans un but universitaire (Eiffel, Objective C, Loops, etc.)[2].

Un objet est caractérisé, dans un système logiciel, par une identité et des propriétés. Les propriétés de l'objet sont définies par son type et peuvent être découpées en deux parties distinctes : l'état et le comportement.

- *l'identité ou OID* (Object Identifier) : l'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. L'OID permet à un objet d'être référencé par d'autres de façon unique et constante pendant toute sa durée de vie ;
- *les attributs* : il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations d'*état* de l'objet ;
- *les méthodes* : les méthodes d'un objet caractérisent son *comportement*, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

La *classe* est la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc "issu" d'une classe. En réalité on dit qu'un objet est une instanciation d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence).
Une classe est composée de deux mêmes parties qu'un objet : ses attributs et ses méthodes qui définissent l'état et le comportement des objets.

Deux instanciations de classes peuvent avoir tous leurs attributs égaux sans pour autant être un seul et même objet. C'est le cas dans le monde réèl, deux T-shirts peuvent

être strictement identiques et pourtant ils sont distincts. D'ailleurs en les mélangeant il serait impossible de les distinguer...

Parmi ces langages orientés objets nous pouvons par exemple citer le C++ ou le Java. L'avantage de ces langages est qu'ils encouragent la modularité. L'interface et l'implémentation des modules peuvent, par exemple, être complètement séparées, ce qui va permettre une certaine réutilisabilité, l'interface cachant les détails inutiles de l'implémentation. En effet, des fragments de code développés dans un cadre différent pourront être réutilisés dans un autre programme.

D'un point de vue pédagogique le grand inconvénient de cette modularité est qu'un grand nombre de modules existent déjà tout faits. L'étudiant ne prend donc pas la peine de chercher ce qui se cache réellement derrière ces codes tout faits, et se contente de les utiliser. Beaucoup de concepts cruciaux de programmation peuvent ainsi lui échapper.

Cette approche favorise donc la possibilité pour l'étudiant de pouvoir écrire rapidement des programmes plus complexes, mais laisse de côté tous les aspects concernant la compréhension plus en profondeur de ce qui se passe réellement au niveau des concepts de programmation.

1.2.2 Approche fonctionnelle

Dans l'approche fonctionnelle [3] le concept clé est, comme son nom l'indique, la fonction. On dit qu'un langage est fonctionnel s'il s'affranchit de façon radicale des effets de bords et interdit toute opération d'assignation[4]. Un programme est alors une application, au sens mathématique du terme, qui donne le même résultat à chaque exécution pour un même ensemble de valeurs en entrée. C'est là que réside la différence fondamentale avec les langages impératifs, tels que le C, dans lesquels des effets de bords sont possibles.

D'autres concepts sont intensément utilisés en programmation fonctionnelle, tels que la récursion ou l'usage des fonctions d'ordre supérieur, c'est-à-dire des fonctions qui peuvent prendre d'autres fonctions comme arguments et/ou retourner une fonction comme résultat.

Pour des raisons d'efficacité et parce que certains algorithmes s'expriment plus facilement avec une machine à états, certains langages fonctionnels autorisent en pratique la programmation impérative en permettant de spécifier que certaines variables sont assignables. La possibilité d'introduire localement des effets de bords est alors bien présente. Les langages dits purement fonctionnels n'autorisent pas la programmation impérative. De ce fait, ils sont dénués d'effets de bords et protégés contre les problèmes que pose l'exécution concurrente.

D'un point de vue pédagogique, les avantages de la programmation fonctionnelle sont multiples. Premièrement, le langage est simple, la syntaxe est facile, et des "recettes de cuisine" toute faites ne sont pas nécessaires. Deuxièmement, les concepts de procédures et fonctions se trouvent au premier plan. Et troisièmement, les programmes sont aisés à comprendre et ne souffrent pas de bogues d'exécution dus à une zone mémoire mal allouée.

Le grand inconvénient de la programmation fonctionnelle est le fait qu'elle est très peu utilisée en entreprise. De ce fait, elle reste fort méconnue au détriment d'autres langages tels que Java et C++.

1.2.3 Approche micromondes

Les micromondes [5] appartiennent au cadre des réflexions sur les méthodes de pédagogie active, et ils font partie des objets d'étude des environnements informatiques pour l'apprentissage humain (EIAH). Un micromonde est le nom donné à un environnement informatique particulier, dans lequel l'utilisateur, et plus précisément, l'étudiant, a une certaine autonomie.

Une approche bien connue basée sur les micromondes est le langage de programmation **Logo** [6] [7] . Logo, un dialecte de Lisp, est un langage de programmation fonctionnel. Il a été développé en 1966 en tant qu'outil pour apprendre. Ses principales caractéristiques sont l'interactivité, la modularité, l'extensibilité et la flexibilité des types de données, ce qui en a fait un langage adapté à l'enseignement des concepts de l'algorithmique. De nos jours, Logo est principalement connu grâce à son "Turtle Graphics". C'est d'ailleurs de cette partie-la de Logo que nous nous sommes inspirés pour le départ de notre implémentation.

L'idée principale du Turtle Graphics est qu'une tortue à laquelle un crayon a été attaché répond à des commandes simples relatives à sa propre position, telles que "MOVE 100" qui lui demande d'avancer de 100 points, ou "LEFT 90" qui demande de se tourner vers la gauche de 90°. La simplicité de ces commandes permet à l'utilisateur de se mettre à la place de la tortue et d'imaginer l'effet qu'aura la commande envoyée. A partir de ces commandes de base, il est possible de composer des figures plus complexes telles des triangles, des cercles, des maisons, ou même des fractales. En créant ces figures l'utilisateur acquiert sans réellement s'en apercevoir toute une série de concepts très importants. Tout l'intérêt du Logo Turtle Graphics est là : permettre d'apprendre sans s'en rendre compte, tout en s'amusant.

Logo est un langage interprété qui possède aujourd'hui plus de 130 variantes. Il forme un compromis entre la programmation séquentielle et la programmation fonctionnelle [9]. Il n'existe néanmoins pas de standard Logo.

Initialement conçu à l'adresse de publics jeunes ou novices en informatique, il est aujourd'hui l'objet de nombreuses contributions qui dépassent largement le cadre de l'éducation à proprement parler.

Une autre approche sur les micromondes est Squeak [10], descendant lointain de Logo, développé autour du langage Smalltalk [11]. Squeak s'inscrit dans les cadres théoriques de la psychologie du développement. L'environnement Squeak est basé sur une machine virtuelle, ce qui assure une large portabilité des programmes.

L'approche basée sur des micromondes tels que Logo et Squeak est une approche plutôt ludique. La syntaxe est assez simple à comprendre et l'environnement est 'user friendly'. L'utilisateur n'a pas vraiment l'impression de programmer mais plutôt de jouer.

Le problème de cette approche est que l'utilisateur est limité dans ses possibilités. En effet, le micromonde est un monde fini. Une fois toutes les fonctionnalités et tous les concepts explorés, il ne reste plus rien de neuf à découvrir. L'utilisateur aurait alors besoin de pouvoir ajouter une nouvelle "couche" à ce monde afin qu'il puisse continuer à apprendre et à découvrir de nouvelles choses. L'interactivité et l'extensibilité, c'est-à-dire la possibilité d'enrichir le langage de base et de travailler sur des couches successives (on pourrait penser à la métaphore des couches d'un oignon) sont des caractéristiques importantes. C'est cet aspect de monde en couches que nous avons développé pour ce mémoire : une fois que l'utilisateur a exploré toutes les possibilités dans le micromonde dans lequel il se trouve, il passe dans un nouveau micromonde contenant toute une nouvelle série de fonctionnalités qui accompagnent les anciennes.

Chapitre 2

Approche par Enrichissement des Micromondes

D'un point de vue pédagogique, l'approche qui semble aujourd'hui susciter le plus grand intérêt est malgré toutes ses limites l'approche des micromondes. La méthode pédagogique que nous avons choisie se base donc sur celle des micromondes.

Un micromonde est un univers très simplifié, dans lequel l'utilisateur a une certaine autonomie alors que ses possibilités sont limitées, justement à cause de cette simplification. Dans chaque micromonde un certain nombre de concepts sont présents et peuvent être utilisés. Les limitations sont volontaires afin de permettre à l'utilisateur de maîtriser un micromonde avant de passer au suivant. Les micromondes font aujourd'hui l'objet de nombreuses réflexions du point de vue de la pédagogie de l'informatique. Les micromondes ont été et sont toujours, d'une certaine manière, le symbole dans le monde informatique des théories de l'éducation nouvelle, c'est-à-dire une éducation qui défend le principe d'une participation active des individus à leur propre formation.

Nous avons commencé par développer un micromonde initial très limité. Nous l'avons enrichi de manière progressive en y ajoutant toujours plus de concepts et de possibilités. Ainsi les micromondes peuvent évoluer au fur et à mesure de l'avancement de l'étudiant. Notre approche est constituée de différents micromondes, chacun reprenant des concepts clés de la programmation. Afin de rendre les micromondes plus évolutifs, nous puiserons dans les avantages des différentes autres approches d'enseignements existantes aujourd'hui que nous détaillerons au chapitre suivant.

Notre approche se focalise sur les concepts de la programmation et non sur un langage bien précis. Afin de rendre possible l'enrichissement des micromondes, il est tout de même nécessaire d'utiliser un langage qui permet l'utilisation des différents paradigmes de programmation, à savoir le style fonctionnel, impératif, orienté objets et orienté agents.

Le monde est fait d'agents et ces agents sont concurrents. Les agents seront donc un concept fondamental dans notre approche. Il n'existe pas une seule et unique définition acceptée en unanimité pour la notion d'agent. Une bonne définition, du point de vue de

notre approche, serait qu'un agent est une entité qui fonctionne continuellement et de manière autonome dans un environnement ou d'autres processus se déroulent et d'autres agents existent[1]. Un agent est donc une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents[2].

Nous avons choisi le langage Oz qui semble bien répondre à ces différents besoins. En effet, celui-ci reprend l'ensemble de ces différents paradigmes. La force de nos micromondes résulte dans le fait que dès le départ, l'étudiant programme en langage Oz, sans forcément s'en rendre compte.

Le processus évolutif mis sur pied va donc permettre à l'étudiant d'avancer par étapes. Une fois qu'une couche a entièrement été assimilée, il pourra passer à la suivante.

2.1 Séquencement des micromondes

Le cours que nous avons développé est divisé en plusieurs micromondes. D'autres micromondes peuvent bien entendu être ajoutés par la suite. Chaque micromonde est un enrichissement du micromonde précédent, les nouveaux concepts d'un nouveau micromonde viennent s'ajouter aux concepts des micromondes précédents qui sont donc conservés.

Commençons par un petit aperçu des différents micromondes utilisés. Ensuite, nous les détaillerons un à un.

Premier micromonde

Le tout premier micromonde est le monde tel qu'il l'était dans le Turtle Graphics de Logo. Nous y abordons le concept d'agents ainsi que la prise en main du logiciel et des différents **agents** inclus à l'environnement : l'agent *Browser* et l'agent *Turtle*. Nous introduisons aussi des concepts de programmation très simples comme les variables et les instructions. Ce micromonde nous semble essentiel avant de pouvoir passer à la suite du cours car il expose quelques éléments de base de la programmation. Sans ce monde, malgré sa simplicité apparente, les étudiants risqueraient d'être perdus pour la suite.

Second micromonde

Dans le second micromonde, les étudiants découvrent une série de concepts plus complexes, dont les principaux sont les **procédures** et la **récursion**. Les étudiants peuvent se familiariser avec ces nouveaux éléments grâce à plusieurs exercices mélangeant les différentes possibilités disponibles à ce stade. Une nouvelle limite est néanmoins rapidement atteinte et de nouveaux concepts seront nécessaires pour pouvoir évoluer.

[1] Cette définition à été proposée par Shoham en 1993 [12]
[2] Cette définition a été proposée par Ferber en 1995 [12]

Troisième micromonde

Dans le troisième micromonde, nous avons principalement introduit le concept de **multi-agents** et donc de **concurrence**. Bien que les agents soient déjà présents dès la première couche, ce n'est que dans celle-ci que les étudiants ont l'occasion de comprendre plus en détail de quoi il s'agit. Ils découvriront de quoi est composé un agent et comment il fonctionne.

Quatrième micromonde

Dans le quatrième micromonde, nous avons ajouté un concept un peu différent, à savoir le **temps**. Ce concept met en évidence la concurrence ainsi que la communication entre différents agents.

Cinquième micromonde

Le cinquième micromonde offre la possibilité à l'étudiant de programmer lui-même le comportement d'un agent. Les étudiants apprennent donc à créer eux-mêmes leurs **propres agents**. Nous leur donnerons également des explications concernant quelques éléments d'**interface graphique** et comment les utiliser.

Nous nous sommes arrêtés à ce cinquième micromonde. Nous abordons ensuite quelques éléments concernant les prochains micromondes qui pourront être développés par la suite :

Sixième micromonde

Une sixième couche, très importante elle aussi, pourra être ajoutée par la suite. Il s'agit de la couche de **multi-component**. Nous n'avons pas eu le temps de développer plus en profondeur cette couche, mais il s'agit de la suite logique du cours. Nous consacrons d'ailleurs une section de ce chapitre à une réflexion sur ce qui pourrait être fait dans le futur.

Septième micromonde

Un septième micromonde pourra également être ajouté par la suite. Ce micromonde abordera la **tolérance aux pannes**. Il pourrait exister un agent dont le rôle est de réagir en cas de panne. Cet agent recevrait alors un message d'un autre agent et aurait pour rôle de réagir en conséquent.

Cette tolérance aux pannes augmenterait la robustesse de l'environnement.

Liens entre les séances de cours et les micromondes

Les séances de cours que nous avons réalisées s'appuient sur les différents micromondes. Voici comment ces séances sont organisées :

1. micromonde 1 : séance de cours 1 - premiers pas : notions de base et prise en main ;
2. micromonde 2 : séance de cours 2 - procédures : séance de cours 3 - récursion ;
3. micromonde 3 : séance de cours 4 - les agents : multi-turtles, concurrence ;
4. micromonde 4 : séance de cours 5 - le temps, Metronome, interaction entre agents ;
5. micromonde 5 : séance de cours 6 - multi-agents : création d'agents spécifiques et mini-projet.

2.2 Premier micromonde

Notre premier micromonde se rapproche fort du *Turtle Graphics* de Logo[6]. Le premier cours s'appuie sur ce micromonde et permet aux étudiants d'apprendre les concepts essentiels à la programmation. La première séance de cours est consacrée à ce micromonde.

2.2.1 Concepts abordés

Voici une liste des concepts qui sont abordés dans ce premier micromonde :

– les agents : il s'agit là du tout premier concept enseigné. Nous présentons deux agents intégrés au micromonde :
  → L'agent **Browser** qui permet d'afficher des informations dans le Browser, afficheur intégré au logiciel ;
  → L'agent **Turtle** qui représente la tortue existant par défaut dans ce micromonde et avec laquelle l'étudiant pourra interagir.
  Les mondes étant composés d'agents, nous estimons que ce concept est central au cours, et cela justifie le fait que nous avons choisi de le présenter en premier.
– variables
– instructions

2.2.2 LogOz

Nous avons tout d'abord commencé par réimplémenter le Turtle Graphics de Logo. Le programme consiste, comme dans la version originale, en une tortue à laquelle un crayon a été attaché. Cette tortue est vue comme un agent qui se trouve dans un micromonde.

Dans un premier temps, le micromonde LogOz ne contient que deux agents : **l'agent Turtle** et **l'agent Browser**. Ces agents sont présents dans le monde lors de son lancement. De cette façon, l'utilisateur pourra immédiatement commencer à interagir avec l'agent Turtle et afficher des données dans le Browser, sans devoir les créer au préalable.

D'autres agents pourront bien évidemment être ajoutés au micromonde par la suite, lorsque l'étudiant aura acquis les connaissances nécessaires. En introduisant ces deux agents nous avons voulu simplifier la prise en main du logiciel et les premiers pas dans la programmation pour un utilisateur débutant.

L'agent Browser est, comme son nom l'indique, un agent qui s'occupera exclusivement de l'affichage. L'utilisateur peut envoyer un message du style

$$\{Send\ Browser\ message\}$$

afin que le Browser affiche le message voulu à un endroit de l'interface prévu à cet effet.

L'agent Turtle est une tortue affichée à l'écran au centre du monde. Il est possible de lui envoyer des commandes simples. Tous ces messages ont un rapport avec l'état dans lequel l'agent se trouve ainsi que sa position dans le micromonde. Les différents messages qui peuvent être reçus par la tortue sont, par exemple, avance de 100 pixels, baisse ton crayon ou tourne vers la gauche selon un angle de 45°.

$$\{Send\ Turtle\ message\}$$

Liste des messages compris par l'agent Turtle

$\{Send\ Turtle\ forward(X)\}$	demande à la tortue d'avancer de X pixel(s)
$\{Send\ Turtle\ penUp\}$	demande à la tortue de lever son crayon
$\{Send\ Turtle\ penDown\}$	demande à la tortue de baisser son crayon
$\{Send\ Turtle\ left(X)\}$	demande à la tortue de tourner vers la gauche d'un angle de X degrés ($0 \leq X \leq 360$)
$\{Send\ Turtle\ right(X)\}$	demande à la tortue de tourner vers la droite d'un angle de X degrés ($0 \leq X \leq 360$)
$\{Send\ Turtle\ color(C)\}$	demande à la tortue de changer la couleur de son crayon (C = green, red, blue, yellow, black, purple, pink)
$\{Send\ Turtle\ name(N)\}$	change le nom de la tortue en N
$\{Send\ Turtle\ show\}$	affiche dans le browser l'état de la tortue
$\{Send\ Turtle\ giveYourState\,(X)\}$	lie la variable X à l'état de la tortue
$\{Send\ Turtle\ kill\}$	détruit la tortue. L'agent, donc le thread, n'existe plus

Tab. 2.1 – Syntaxe des messages

Si l'agent reçoit un message qu'il ne comprend pas, il envoit lui même un autre message à l'agent Browser lui demandant d'afficher qu'il ne comprend pas le message qu'il vient de recevoir.

Diagramme d'état de l'agent Turtle

La figure 2.1 donne le diagramme d'état de l'agent *Turtle*.

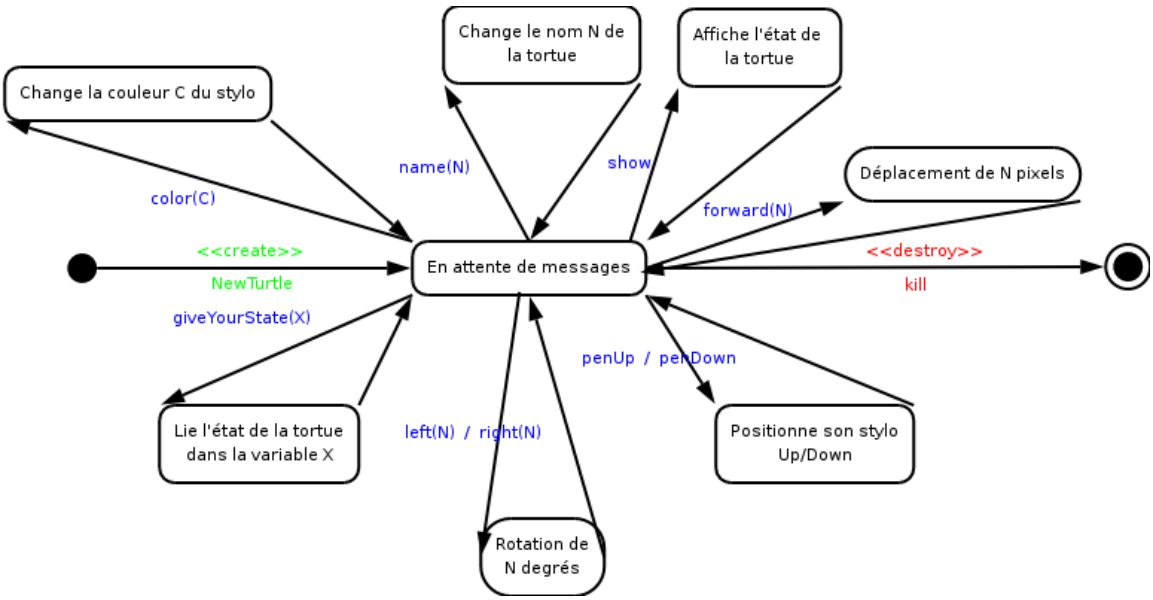

FIG. 2.1 – Diagramme d'état de l'agent Turtle

Géométrie de la tortue

La tortue peut être considérée comme un crayon qui se déplace sur l'écran et qui peut être contrôlé par l'utilisateur à l'aide d'un ensemble d'instructions (cf Tab. 2.1). La tortue évolue dans un monde représenté par un carré de X*X pixels. La tortue se trouve initialement au milieu du carré et est orientée vers la droite (l'angle 0 en trigonométrie). Nous avons conservé la notation traditionnelle en trigonométrie afin de rester cohérents dans nos calculs.

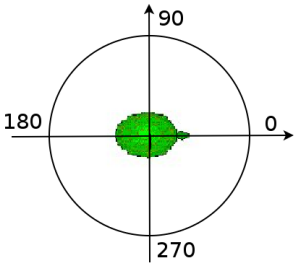

FIG. 2.2 – Orientation de la tortue

La tortue peut effectuer des rotations de 0° à 360° vers la gauche, ou vers la droite. Des rotations d'angles plus grands sont également possibles, mais sans intérêt. Le monde de la tortue est délimité par deux axes, x et y. Le point (0,0) se trouve en haut à gauche

du monde.

Exemple

Grâce aux messages, nous pouvons par exemple demander à la Turtle de tracer un triangle :

```
X = 100
{Send Turtle penDown}
{Send Turtle forward(X)}
{Send Turtle left(120)}
{Send Turtle forward(X)}
{Send Turtle left(120)}
{Send Turtle forward(X)}
{Send Turtle left(120)}
```

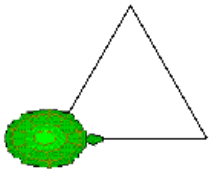

FIG. 2.3 – un triangle

2.2.3 Conclusion sur le premier micromonde

Ce premier micromonde est simple, il permet à l'étudiant de se familiariser avec l'environnement et la syntaxe des messages. L'utilisateur est cependant limité à ces messages qui sont définis au préalable, tout comme il l'était dans la version du Turtle Graphics de Logo. Une évolution vers un second micromonde est nécessaire à ce stade. Après avoir exploré les commandes de base et compris les concepts de la première partie, l'étudiant aura épuisé les possibilités de ce premier micromonde. Il aura alors les connaissances basiques essentielles pour pouvoir passer à l'étape suivante.

2.3 Second micromonde

Notre second micromonde est une évolution du premier enrichi essentiellement par les concepts de **procédure** et de **récursion**. Les séances de cours 2 et 3 y sont consacrées. La séance 2 se focalise essentiellement sur le concept de procédure et la séance 3 sur la récursion.

C'est donc ici qu'apparaît la différence entre notre nouvelle version du Turtle Graphics et la version de Logo : alors que les possibilités du Turtle Graphics étaient limitées -une fois que l'utilisateur avait exploré toutes les possibilités, il ne lui restait plus rien à découvrir-, notre version a pour objectif d'évoluer. A l'aide du langage de programmation Oz[13] que nous avons utilisé pour l'implémentation, nous avons créé de nouvelles couches qui ajoutent de nouvelles possibilités dès que l'étudiant n'a plus rien de neuf à explorer.

De plus, l'utilisateur ne se sert d'aucun sous-langage lorsqu'il programme, il utilise le Oz directement à travers l'interface qui lui est proposée. Le fait d'utiliser ainsi un réel langage

de programmation permet d'offrir rapidement de nombreuses possibilités d'évolution dans notre Oz Turtle Graphics.

2.3.1 Concepts abordés

Voici une liste des concepts qui sont abordés dans ce second micromonde :

- procédure, arguments ;
- portée ;
- les nombres entiers (Integer) et les nombres à virgule (Float) ;
- les opérateurs arithmétiques (+, -, *, div, /, mod) ;
- les opérateurs binaires (==, \ =, =<, <, >=, >) ;
- conditionnelle : if, pattern matching : case of ;
- la récursion.

2.3.2 Procédures avec et sans arguments

Une fois que les commandes de base ont convenablement été comprises par l'étudiant, le concept de procédure est très vite introduit. La procédure est tout d'abord présentée de manière simple, c'est-à-dire sans aucun argument. Une procédure est un sous-programme qui permet d'effectuer un ensemble d'instructions par un simple appel de la procédure. La procédure peut être vue comme une suite d'instructions à laquelle on a donné un nom, un peu comme une recette de cuisine. L'accent est plutôt mis sur la grande utilité du concept ainsi que sur les facilités qu'il apporte.

Dans un second temps, les arguments sont ajoutés aux procédures, ce qui rend ce concept encore plus intéressant.

Voici un exemple d'utilisation de procédures avec argument :

```
proc{Triangle X}
   {Send Turtle forward(X)}
   {Send Turtle left(120)}
   {Send Turtle forward(X)}
   {Send Turtle left(120)}
   {Send Turtle forward(X)}
   {Send Turtle left(120)}
end
{Send Turtle penDown}
{Triangle 50}
{Triangle 100}
{Triangle 150}
```

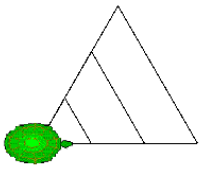

FIG. 2.4 – Une procédure pour dessiner un triangle avec 1 argument

2.3.3 Répétition

La répétition est ensuite introduite à l'aide de la procédure *Repeat* :

$$\{Repeat\ N\ P\}$$

Ce concept va de pair avec les procédures, du moins dans l'idée de faire du dessin. La répétition dans notre micromonde n'est rien d'autre qu'une procédure qui va répéter N fois une autre procédure P.

La procédure Repeat a été ajoutée dans l'environnement pour le deuxième micromonde, elle peut donc être utilisée telle quelle. Des grandes possibilités s'offrent alors aux étudiants : les procédures peuvent appeler d'autres procédures, elles peuvent prendre d'autres procédures en arguments, et ainsi de suite.

Voici deux exemples d'utilisation de la procédure *Repeat* :

```
proc{Cercle}
   {Send Turtle forward(1)}
   {Send Turtle left(1)}
end
{Send Turtle penDown}
{Repeat 360 Cercle}
```

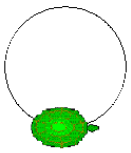

FIG. 2.5 – Première façon de dessiner un cercle

```
{Send Turtle penDown}
{Repeat 360 proc{$}
       {Send Turtle forward(1)}
       {Send Turtle left(1)}
end}
```

FIG. 2.6 – Deuxième façon de dessiner un cercle

Les étudiants pourront rapidement faire de jolis dessins en utilisant bon nombre de concepts étudiés.

Dessiner une fleur

Voici un autre exemple d'utilisation de procédures. Plusieurs procédures vont être imbriquées afin de réaliser le dessin d'une fleur (Fig. 2.7) :

```
proc{QCercle}
   {Repeat 45 proc{$} {Send Turtle forward(2)} {Send Turtle left(2)} end}
end
proc{Petale}
   {Repeat 2 proc{$} {QCercle} {Send Turtle left(90)} end}
end
proc{Fleur}
   {Repeat 10 proc{$} {Petale} {Send Turtle right(36)} end}
end
proc{Plante}
   {Send Turtle color(red)}
   {Fleur}
   {Send Turtle color(green)}
   {Send Turtle left(180)}
   {Send Turtle forward(130)}
   {Send Turtle left(90)}
   {Petale}
   {Send Turtle right(90)}
   {Send Turtle forward(70)}
end
{Send Turtle penDown} {Send Turtle left(90)}
{Plante}
```

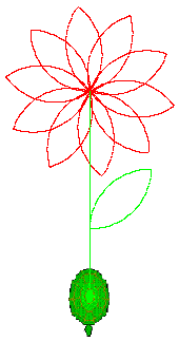

FIG. 2.7 – Une fleur

La répétition et les procédures sont les concepts les plus importants étudiés dans la seconde séance de cours.

2.3.4 Récursion et opérateurs

Un autre concept essentiel introduit dans la troisième séance de cours est la récursion. Celle-ci permet de réaliser des boucles et des dessins plus complexes. Afin de pouvoir introduire ce concept, il est d'abord nécessaire d'apprendre aux étudiants les opérateurs de base, les structures conditionnelles ainsi que le pattern matching. Nous ne nous attardons

pas ici sur ces différents concepts, ceux-ci se trouvent dans la séance de cours 3 dans l'annexe A.

Une fois ces opérateurs de base, les structures conditionnelles et le pattern matching bien maîtrisés, nous pouvons enfin nous attaquer à la récursion. Ici non plus les exemples explicites ne manquent pas ! Nous avons choisi comme fil conducteur pour la récursion l'exemple classique de l'arbre (Fig. 2.8).

```
proc{Tree Size Depth}
   if(Depth == 0) then skip
   else
      {Send Turtle forward(Size)}
      {Send Turtle left(20)}
      {Tree Size div 2 Depth-1}
      {Send Turtle right(40)}
      {Tree Size div 2 Depth-1}
      {Send Turtle left(20)}
      {Send Turtle left(180)}
      {Send Turtle forward(Size)}
      {Send Turtle left(180)}
   end
end
{Send Turtle left(90)}
{Send Turtle penDown}
{Tree 200 5}
```

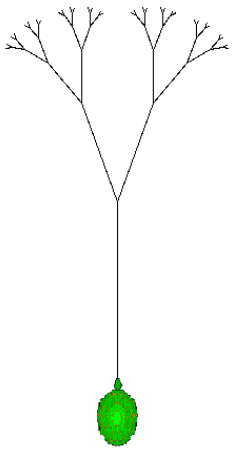

FIG. 2.8 – Un arbre

Il existe un nombre impressionnant d'exemples possibles de dessins utilisant la récursion. Nous avons réalisé quelques "chefs-d'oeuvre" que nous ne présentons pas ici mais qui se trouvent dans l'annexe B. Ces exemples pourront également être utilisés comme exercices dans le cadre de travaux pratiques.

A ce stade du cours, les étudiants ont suffisamment de connaissances pour pouvoir reprogrammer eux-mêmes la procédure *Repeat*. Il n'est donc plus nécessaire de leur cacher son implémentation. Ceci est en quelque sorte un aboutissement de la seconde couche. Voici une implémentation différente de Repeat, *MyRepeat* (Fig. 2.9) :

```
proc{MyRepeat N P}
   if N==0 then skip
   else {P}
        {MyRepeat N-1 P}
   end
end
```

FIG. 2.9 – Procédure Repeat

2.3.5 Conclusion du second micromonde

Les possibilités qui sont offertes à ce stade sont réellement énormes. Les outils dont les étudiants disposent maintenant sont pourtant très simples et les concepts restent assez basiques. Nous pouvons donc observer qu'il ne faut pas toujours chercher à cacher les difficultés de la programmation aux étudiants. Parfois, il suffit de prendre le temps d'expliquer certaines choses pour que par la suite il y ait une meilleure compréhension de l'ensemble des concepts.

L'exemple à la figure 2.10 reprend à peu près tous les concepts vus jusqu'à présent, mais surtout les procédures et la récursion.

Dessiner une fractale [14]

```
proc{Triangle N}
   if(N<4) then skip
   else
      {Repeat 3 proc{$} {Send Turtle forward(N)} {Send Turtle right(120)} end}
      {Triangle (N div 2)}
      {Send Turtle penUp}
      {Send Turtle forward(N div 2)}
      {Send Turtle penDown}
      {Triangle (N div 2)}
      {Send Turtle penUp}
      {Send Turtle right(120)}
      {Send Turtle forward(N div 2)}
      {Send Turtle left(120)}
      {Send Turtle penDown}
      {Triangle (N div 2)}
      {Send Turtle penUp}
      {Send Turtle right(240)}
      {Send Turtle forward(N div 2)}
      {Send Turtle right(120)}
      {Send Turtle penDown}
   end
end

{Send Turtle left(45)}
{Send Turtle penDown}
{Triangle 384}
```

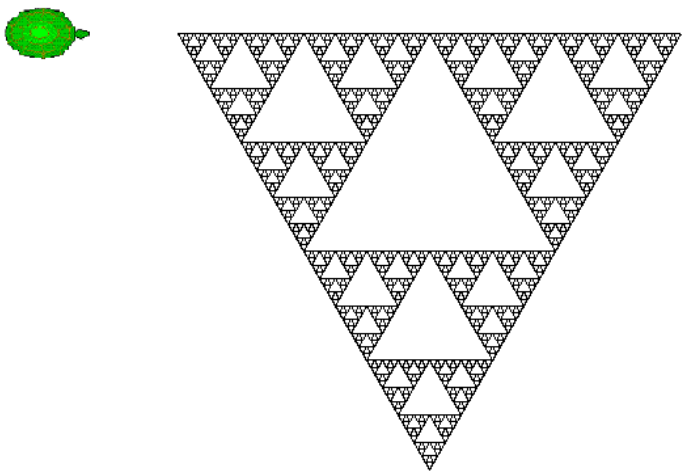

FIG. 2.10 – Exemple de fractale

2.4 Troisième micromonde

Notre troisième micromonde est une évolution du second enrichi essentiellement par le concept de **multi-agents**. Dans ce micromonde, l'étudiant va pouvoir créer et utiliser plusieurs agents de type *Turtle*. Le cours 4 s'appuie sur ce micromonde.

2.4.1 Concepts abordés

Voici une liste des concepts abordés dans ce troisième micromonde :
- fonctions ;
- l'état d'un agent ;
- tuples ;
- fonction de transition d'un agent ;
- conversion entre entiers (Integer) et nombres réels (Float) ;
- la concurrence.

2.4.2 Les fonctions

Une **fonction** est présentée comme une procédure dont la différence fondamentale est qu'elle renvoie toujours un résultat (une valeur) alors que la procédure non. Nous ne présentons pas ici les détails concernant les fonctions. Ceux-ci se trouvent dans le cours 4 en annexe A. Afin d'illustrer ce concept, le code de la figure 2.11 présente la fonction *RandomColor* permettant de renvoyer une couleur de manière aléatoire. Cette fonction utilise une également une fonction *Random* qui a été ajouté à ce micromonde et qui renvoie un nombre entier aléatoire positif.

```
fun{RandomColor}
   Rand = {Random} mod 9 in
   if(Rand == 0) then black
   elseif(Rand == 1) then blue
   elseif(Rand == 2) then green
   elseif(Rand == 3) then red
   elseif(Rand == 4) then pink
   elseif(Rand == 5) then purple
   elseif(Rand == 6) then orange
   elseif(Rand == 7) then brown
   else yellow end
end
```

FIG. 2.11 – Fonction RandomColor

En utilisant cette fonction pour dessiner la fractale, on peut obtenir le dessin de la figure 2.12.

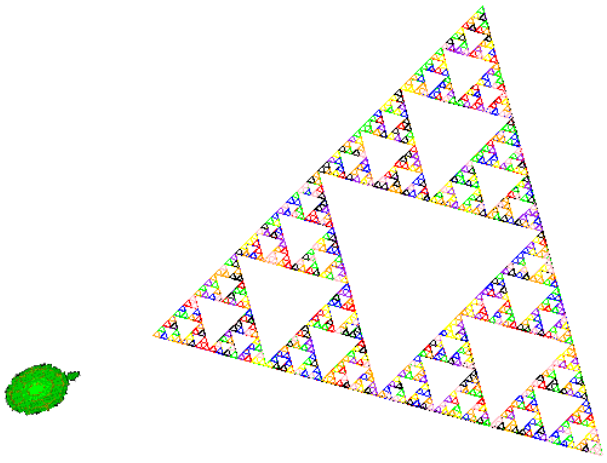

FIG. 2.12 – Fractale en couleur

2.4.3 Sémantique de la tortue

La tortue (et plus généralement les agents) est un *port object* qui utilise la technique du *message passing*[15]. Un port object est une combinaison de un ou plusieurs *ports* et un *stream object*. La tortue est implémentée comme un thread qui lit les messages qui lui

parviennent via le stream object[3]. Toute la communication avec la tortue se fera donc via ce stream object. Chaque message valide reçu par la tortue entraînera une action qui est déterminée à l'avance.

La tortue étant un port object et non pas un stream object seulement, elle peut recevoir des messages venant de différents expéditeurs. Cet aspect de la communication sera très important lorsque plusieurs agents interagiront dans le micromonde.

L'implémentation de la tortue comme un port object est dans un premier temps bien évidemment cachée aux étudiants. Ils la découvriront par la suite, lorsqu'ils auront accumulé assez de connaissances pour en comprendre le fonctionnement.

Etat de la tortue

La tortue possède un état interne qui est mis à jour à la réception de chaque nouveau message. Cet état est un peu comme sa mémoire. Un étudiant peut visualiser l'état de la tortue en cliquant sur celle-ci. Il s'affichera alors dans le Browser intégré de l'interface.

Voici l'état interne de la tortue qui consiste en un tuple :

$$state(Name\ X\ Y\ Direction\ PenDown\ Color)$$

- $Name$: correspond tout simplement au nom de l'agent ;
- X *et* Y : correspondent aux coordonnées X et Y de la tortue dans le monde (Rappel : le point (0,0) se trouve en haut à gauche du monde !) ;
- $Direction$: correspond à la direction (angle trigonométrique) vers laquelle la tortue est orientée ;
- $PenDown$: booléen qui permet de savoir si le crayon est baissé (true) ou non (false) ;
- $Color$: indique la couleur dans laquelle les traits qui sont tracés apparaissent à l'écran.

L'état d'un agent étant un tuple, nous introduisons ici le concept de *tuple*. Nous savons bien qu'il a déjà été utilisé précédemment dans les messages qui peuvent être envoyés à la tortue, mais nous estimons qu'il n'est pas vraiment nécessaire d'expliquer les tuples avant le troisième cours.

L'étudiant se familiarise également avec le concept de *variable liée et non liée*, tout en approfondissant sa compréhension de l'état. Dans l'exemple de la figure 2.13 l'étudiant va demander à sa tortue de lier une variable qu'il aura créée à son état :

```
X
{Send Browser X}                % X n'est pas lié
{Delay 2000}
{Send Turtle giveYourState(X)} % X est maintenant lié à l'état
```

FIG. 2.13 – Liaison d'une variable à l'état de la tortue

[3]Le détail de l'implémentation se trouve dans le chapitre 3 au point 3.2 *Implémentation des agents.*

Fonction de transition de la tortue

La fonction de transition de la tortue permet d'accepter seulement quelques messages prédéfinis. Chaque message reçu modifie l'état de la tortue en conséquence. Les messages compris par la tortue sont disponibles dans le Tab. 2.1. La fonction de transition de la tortue ne peut donc pas être modifiée par les étudiants. Dans un micromonde suivant, l'étudiant sera amené à implémenter lui-même des fonctions de transition. L'objectif ici est de permettre à l'étudiant de créer et ensuite d'utiliser plusieurs agents d'une manière aussi simple que possible.

2.4.4 Création d'un nouvel agent tortue

L'étudiant va se familiariser avec la programmation multi-agents en commençant par créer de nouveaux agents *Turtle*. Ces tortues auront un comportement identique (et donc une fonction de transition identique) à la tortue intégrée au micromonde. L'étudiant va commencer par paramétrer l'état de la tortue qu'il crée :

$$state('Mat'\ 440\ 340\ 270\ true\ black)$$

Une fonction *NewTurtle* a été ajoutée dans l'environnement pour ce micromonde, permettant à l'étudiant de créer de nouveaux agents de type *Turtle*. Cette fonction possède comme unique argument l'état dans lequel doit se trouver la tortue que l'on souhaite créer.

Création d'une tortue

```
MyTurtle = {NewTurtle state('Mat' 440 340 270 true black)}
```

L'étudiant pourra ensuite réaliser des programmes dans lesquels interviennent plusieurs tortues. Dans l'exemple présenté aux figures 2.14 et 2.15, chaque tortue apporte sa contribution à la réalisation du dessin :

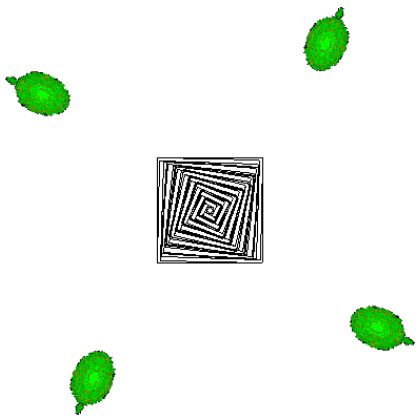

FIG. 2.14 – 4 tortues

```
T1 = {NewTurtle state('t1' 440 340 270 true black)}
T2 = {NewTurtle state('t2' 440 440 180 true black)}
T3 = {NewTurtle state('t3' 340 440 90 true black)}
{Send Turtle penDown}
proc{Carre T X}
   if(X =< 0) then
      {Send T penUp}
      {Send T forward(200)}
      skip
   else
   {Send T forward(X)}
   {Send T right(91)}
   {Carre T X-3}
   end
end

{Carre Turtle 100}
{Carre T1 100}
{Carre T2 100}
{Carre T3 100}
```

FIG. 2.15 – Code des 4 tortues

2.4.5 Plusieurs tortues : la concurrence

Le troisième micromonde est enrichi, comme mentionné précédemment, par une nouvelle couche multi-agents. Comme son nom l'indique, cette couche va permettre à l'utilisateur de programmer avec plusieurs agents. Les différents agents peuvent soit être totalement indépendants, soit interagir entre eux. Dans ce micromonde, l'étudiant pourra seulement créer des agents de type *Turtle*. Chaque agent aura son propre nom ainsi que son propre identificateur, et pourra effectuer ses propres actions. Cette évolution apporte un tout nouveau concept clé : la **concurrence**.

La concurrence apparaît lorsque deux ou plusieurs activités indépendantes peuvent être exécutées simultanément. Il ne devrait y avoir aucune interférence entre ces activités, à moins que le programmeur ne décide qu'il est nécessaire qu'elles communiquent entre elles[15].

Plusieurs agents pourront donc effectuer différentes actions en même temps. La concurrence ne deviendra source de problèmes ou de conflits qu'à partir du moment où les agents devront utiliser des ressources partagées.

2.4.6 Conclusion sur le troisième micromonde

Dans ce troisième micromonde, l'étudiant crée et utilise plusieurs agents. Pour l'instant ces agents sont tous de même type, mais ceci n'est qu'un premier pas vers la programmation multi-agents. Ce micromonde lui aura permis de comprendre le fonctionnement interne d'un agent. L'étudiant possède maintenant le bagage nécessaire pour pouvoir comprendre comment créer ses propres agents.

2.5 Quatrième micromonde

Ce quatrième micromonde est une évolution du troisième enrichi par le concept de **temps**. Dans ce micromonde, l'étudiant va pouvoir approfondir sa compréhension de la concurrence en faisant interagir des agents entre eux.

2.5.1 Concepts abordés

Voici une liste des concepts qui sont abordés dans ce quatrième micromonde :
- le temps
- un autre type d'agent : les **Metronomes**
- les listes

Un autre concept très important est donc étudié dans cette partie : le concept de **temps**. Le principe est simple : un agent que nous avons appelé *Metronome* est intégré au micromonde. Il est chargé d'émettre un message toutes les secondes. Ce message peut être envoyé à d'autres agents si l'étudiant le souhaite. Il lui suffit d'ajouter l'identifiant de l'agent à la liste des agents à qui le Metronome doit envoyer son message. L'étudiant devra déterminer le contenu du message qui est envoyé aux agents. Il doit également s'assurer que le message est bien compris par l'agent qui le reçoit. Dans le cas contraire, l'agent ne comprendra pas le message et n'effectuera donc aucune d'action.

2.5.2 Mise en place d'un agent Metronome

Une fonction *NewMetronome* a été ajoutée à ce quatrième micromonde, permettant de créer un agent de type *Metronome*. Cette fonction prend comme premier argument une liste contenant les agents auxquels il doit envoyer un certain message, et comme second argument le message à envoyer :

```
MyClock = {NewMetronome state([ListOfAgents] MsgToSend)}
```

Ceci nous permet également d'introduire le concept des **listes**, étant donné que les agents de type *Metronome* ont besoin d'une liste d'agents auxquels ils doivent envoyer leur message. Nous ne verrons pas ce concept en détail ici. Les explications sur les listes se trouvent en annexe A.

Messages compris par l'agent Metronone

L'envoi d'un message au Metronome se fait à nouveau sous cette forme :

$$\{Send\ MyClock\ message\}$$

Voici la liste des messages que le Metronome comprend :

start	démarre le Metronome
stop	arrête le Metronome
register(*Agent*)	ajoute l'agent *Agent* à la liste des agents auxquels un message sera envoyé chaque seconde
message(*Msg*)	définit le message *Msg* à envoyer aux agents
clear	réinitialise l'état du Metronome à *state*(*nil nil*) La liste d'agents est vide ainsi que le message à envoyer.

TAB. 2.2 – Syntaxe des messages

Diagramme d'état de l'agent Metronome

La figure 2.16 donne le diagramme d'état d'un agent de type *Metronome*.

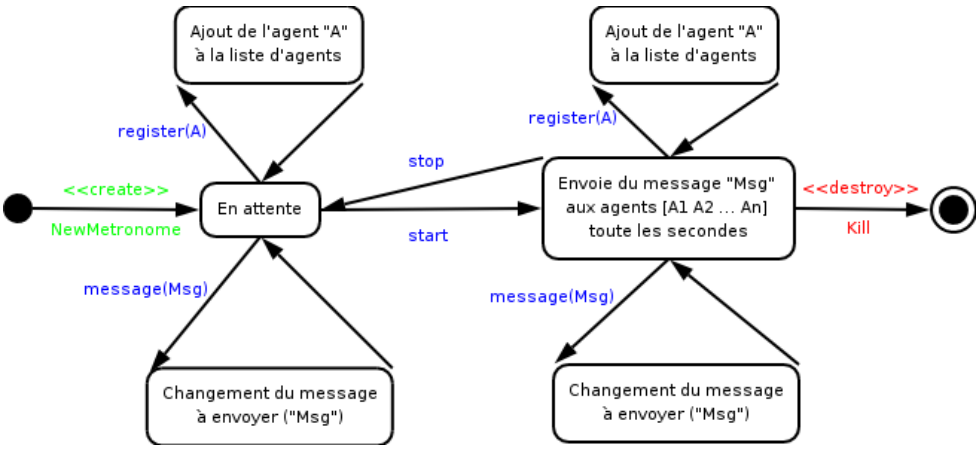

FIG. 2.16 – Diagramme d'état de l'agent Metronome

Ce concept de temps illustre bien la concurrence qu'il peut y avoir entre les différents agents. Si plusieurs agents reçoivent le message au même moment, ils réagiront en même temps. Cela ne pose dans notre cas aucun problème car les agents n'agissent pas sur des ressources partagées. L'utilisateur a simplement l'illusion que les agents s'exécutent en même temps.

L'exemple présenté à la figure 2.17 illustre ce principe de manière toute simple. Le Metronome *MyClock* créé par l'utilisateur va envoyer toutes les secondes un message $forward(100)$ aux deux tortues qu'il a créées.

```
{Send Turtle kill}  % supprime l'agent Turtle
T1 = {NewTurtle state('t1' 440 340 270 true green)}
T2 = {NewTurtle state('t2' 100 200 0 false blue)}
MyClock = {NewMetronome state(nil nil)}
{Send MyClock register(T1)}
{Send MyClock register(T2)}
{Send MyClock message(forward(100))}
{Send MyClock start}
```

FIG. 2.17 – Interaction entre un agent Metronome et deux tortues

2.5.3 Conclusion sur le quatrième micromonde

Ce quatrième micromonde met en évidence la communication entre différents agents. Cet aspect est essentiel avant de passer à la couche suivante qui est la création d'agents dont l'utilisateur programme le comportement.

2.6 Cinquième micromonde

Notre cinquième micromonde est une évolution du quatrième enrichi par des concepts plus avancés concernant le **multi-agents**. Dans ce micromonde, l'étudiant va pouvoir créer ses propres agents en définissant leur comportement à l'aide de fonctions de transition qu'il implémentera. L'étudiant aura également la possibilité de se familiariser avec certains éléments des interfaces graphiques.

2.6.1 Concepts abordés

Voici une liste des concepts qui sont abordés dans ce cinquième micromonde :
- création d'agents
- état et fonction de transition : implémentation
- éléments d'interface graphique

2.6.2 Création d'un agent

Afin de permettre aux étudiants de créer des agents dont ils vont définir eux-mêmes le comportement, une fonction *NewAgent* a été ajoutée dans l'environnement pour ce cinquième micromonde.

```
MyAgent = {NewAgent Fun State}
```

Comme pour la tortue, le port object reste toujours caché pour l'étudiant mais il peut maintenant implémenter une fonction de transition. Il choisit également un état afin de déterminer lui-même le comportement de son agent.

Prototype d'une fonction de transition

La figure 2.18 présente le prototype d'une fonction de transition :

```
fun{FonctionTransition Msg State}
   case Msg
   of ... then
         ....
         %renvoie le nouvel état
   [] ... then
         ...
         %renvoie le nouvel état
   else
         ...
         %renvoie le nouvel état
   end
end
```

FIG. 2.18 – Prototype d'une fonction de transition

Exemple : un compteur

Un premier exercice aura pour but de créer un agent simple, par exemple un compteur. Ce compteur doit s'incrémenter chaque fois qu'il reçoit un message *tick* par exemple. L'étudiant doit implémenter sa fonction de transition. Une fois ce compteur opérationnel, il peut également utiliser un agent Metronome qui se chargera de lui envoyer un message toutes les secondes afin qu'il incrémente son état. Voici une illustration de cette communication entre agents :

L'étudiant aura ensuite l'occasion de mettre en pratique tout ce qu'il a appris en programmant plusieurs agents différents qui communiquent et interagissent entre eux.

Des applications de taille plus importante peuvent déjà être réalisées à ce stade. Les étudiants seront par exemple amenés à réaliser une horloge à aiguilles, ce qui permet d'illustrer la programmation multi-agents. Nous leur donnons des outils leur permettant de gérer eux-mêmes l'interfaçage graphique.

Il existe aujourd'hui un tas d'applications pour des programmes multi-agents, et parmi ceux-ci beaucoup sont issus du domaine des jeux vidéo. Les jeux ont pour avantage d'être des exemples concrets, explicites, motivants et attrayants pour les étudiants. Des jeux

```
% fonction de transition
fun{FTCompteur Msg State}
   case Msg
   of tick then
      {Send Browser State+1} %affichage
      State+1
   else
      {Send Browser 'I do not understand the message '#Msg}
      State
   end
end

% création de l'agent Compteur
Compteur = {NewAgent FTCompteur 0}

% Utilisation de l'agent
{Send Compteur tick}      % affiche '1' dans le Browser
{Send Compteur tick}      % affiche '2'
{Send Compteur tack}      % affiche 'I do not understand the message tack'

% Utilisation d'un Metronome pour envoyer le message
M = {NewMetronome state([Compteur] tick)}
{Send M start}
```

FIG. 2.19 – Implémentation d'un compteur

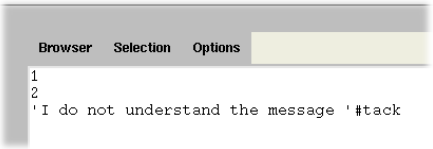

FIG. 2.20 – Un compteur

simples tels que le jeu de ping-pong dans lequel il y a deux palettes et une balle, le but étant de renvoyer la balle à l'adversaire. Le jeu du "Game Of Life" ou encore "Space Invaders" et beaucoup d'autres sont également des applications de la programmation multi-agents.

Ces applications nécessitant une partie graphique, nous avons fourni quelques outils permettant aux étudiants de créer des interfaces simples, qui ne nécessitent pas de connaissances de QTk[4].

[4]Graphical User Interface Design for Oz[16]

2.6.3 Outils d'interfaçage graphique

Afin de réaliser certains exercices prévus dans les cours que nous avons élaborés, nous avons ajouté dans ce micromonde quelques fonctions et procédures permettant de faire un peu de graphisme. A partir de maintenant, les étudiants ne sont plus de simples utilisateurs du micromonde, ils sont également des acteurs et créateurs ! Ils peuvent laisser libre cours à leur imagination et agir sur le micromonde.

DrawLine

$Tag = \{DrawLine\ Size\ Direction\ Color\}$
Cette fonction possède 3 arguments :

1. *Size* qui permet de définir la longueur de la ligne que l'on dessine à l'écran (en pixels) ;
2. *Direction* qui permet d'orienter la ligne que l'on dessine à l'écran (en degrés) ;
3. *Color* qui permet de définir la couleur de la ligne que l'on dessine à l'écran.

Cette fonction renvoie ce que l'on appelle un tag qui permet d'identifier la ligne dessinée. Ce tag va permettre d'effacer une ligne par la suite.

DeleteTag

$\{DeleteTag\ Tag\}$
Cette procédure supprime l'image à l'écran identifiée par le Tag.

DrawBall

$TagBall = \{DrawBall\ X\ Y\}$
Cette fonction dessine à l'écran une balle à la position (X, Y) et renvoie un tag permettant d'identifier cette balle.

DrawPaddle

$TagPaddle = \{DrawPaddle\ X\ Y\}$
Cette fonction dessine à l'écran une palette de taille fixe (100 pixels) à la position (X, Y) qui correspond au haut de la palette, et renvoie également un tag.

MoveTag

$\{MoveTag\ Tag\ X\ Y\}$
Cette procédure déplace le dessin identifié par le tag de sa position horizontale actuelle posX vers posX + X et de sa position verticale actuelle posY vers posY + Y.

2.6.4 Exemple de programme multi-agents

Une Horloge

Nous avons choisi d'implémenter une horloge comme premier exemple de programme multi-agents. L'horloge est composée de 3 agents : l'agent Seconde, l'agent Minute et l'agent Heure. Une fois arrivé à 60, l'agent Seconde enverra un message à l'agent Minute, qui une fois arrivé à 60 enverra un message à l'agent Heure. La figure 2.21 présente le diagramme de séquence de cette horloge, il permet de voir les interactions entre les différents agents. Vient ensuite l'implémentation de cette horloge et pour finir la figure 2.22 présente le résultat.

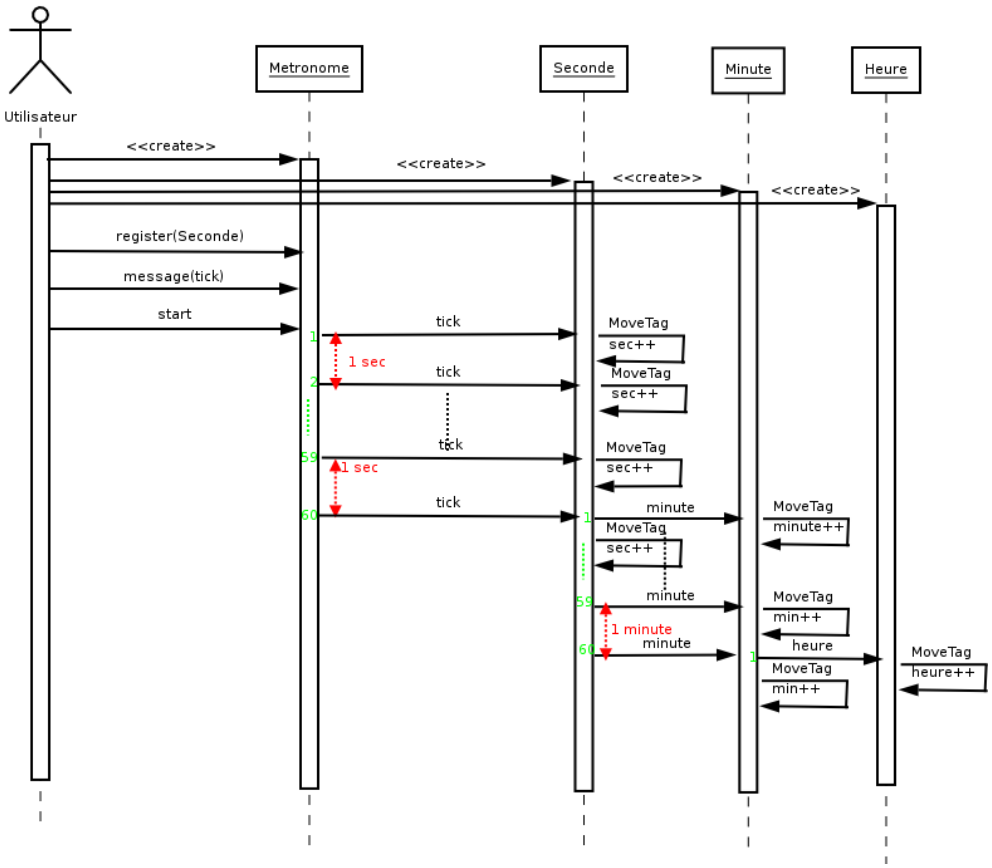

FIG. 2.21 – Diagramme de séquence de l'horloge

```
%%% Fonction de transition de l'agent Seconde
fun{BrainSecond Msg state(Memory X)}
   case Msg
      of tick then
         {DeleteTag X}
         if(Memory > 0) then
            if(Memory == 90) then {Send Minute minute} end
            NewX = {DrawLine 150 Memory-6 red} in
            state(Memory-6 NewX)
         else
            NewX={DrawLine 150 Memory+360-6 red} in
            state(Memory+360-6 NewX)
         end
      [] init(D) then
         X = {DrawLine 150 D red}
         state(D X)
      end
end

%%% Fonction de transition de l'agent Minute
fun{BrainMinute Msg state(Memory X)}
      case Msg
         of minute then
            {DeleteTag X}
            if(Memory > 0) then
               if(Memory mod 24 == 0) then {Send Heure heure} end
               NewX = {DrawLine 140 Memory-6 black} in
               state(Memory-6 NewX)
            else
               NewX = {DrawLine 140 Memory+360-6 black} in
               state(Memory+360-6 NewX)
            end
         [] init(D) then
            X = {DrawLine 140 D black}
            state(D X)
         end
end
```

```
%%% Fonction de transition de l'agent Heure
fun{BrainHour Msg state(Memory X)}
   case Msg
      of heure then
         {DeleteTag X}
         if(Memory > 0) then
            NewX = {DrawLine 130 Memory-6 black} in
            state(Memory-6 NewX)
         else
            NewX = {DrawLine 130 Memory+360-6 black} in
            state(Memory+360-6 NewX)
         end
      [] init(D) then
         X = {DrawLine 130 D black}
         state(D X)
      end
end

%%% Création des agent
Seconde = {NewAgent BrainSecond state(78 _)}
Minute = {NewAgent BrainMinute state(96 _)}
Heure = {NewAgent BrainHour state(6 _)}

{Send Seconde init(78)}
{Send Minute init(96)}
{Send Heure init(6)}
Clock = {NewMetronome state([Seconde] tick)}
% seul l'agent Seconde recevra un TIC à chaque seconde.
% L'agent Minute réagira aux messages envoyés par l'agent Seconde.
% L'agent Heure réagira aux messages envoyés par l'agent Minute.

{Send Clock start}
```

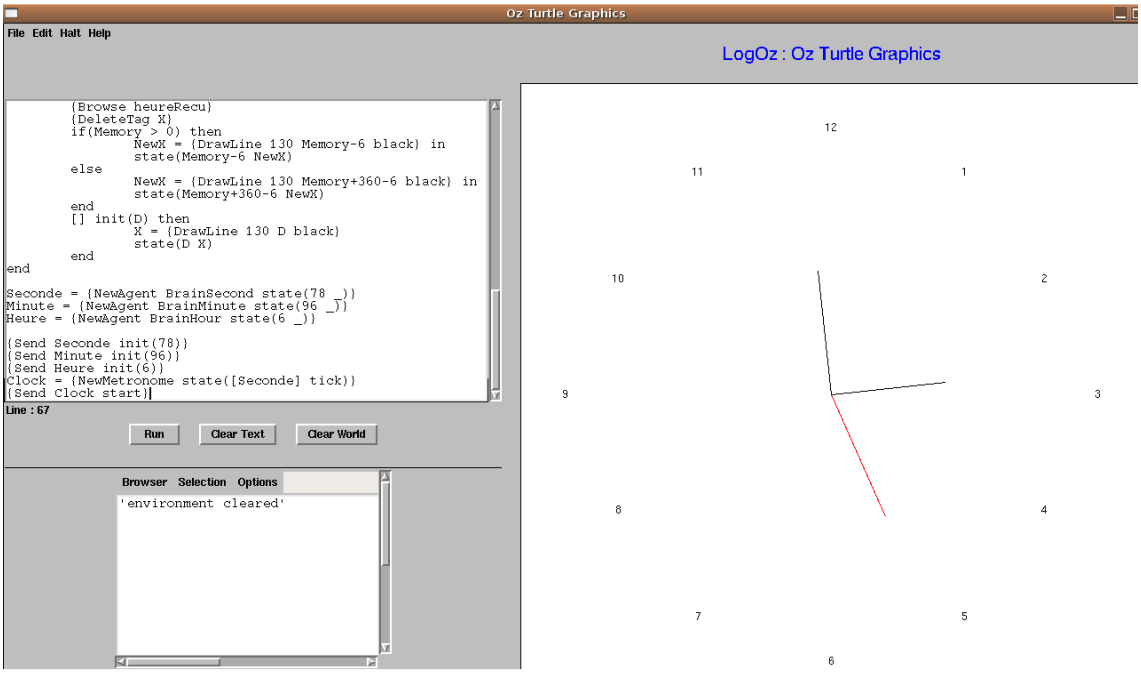

FIG. 2.22 – Une horloge

Un jeu de ping-pong

Un second exemple, plus conséquent, de programme multi-agents est l'implémentation d'un jeu de ping-pong. Ce jeu pourrait servir de premier petit projet pour les étudiants. Nous ne donnons pas ici le code de l'exemple car il prendrait trop de place, mais il se trouve dans le cours 6 en annexe A. Il est accompagné d'une description détaillée sur les choix d'implémentation ainsi que sur le fonctionnement du jeu. La figure 2.23 montre le résultat final de l'implémentation.

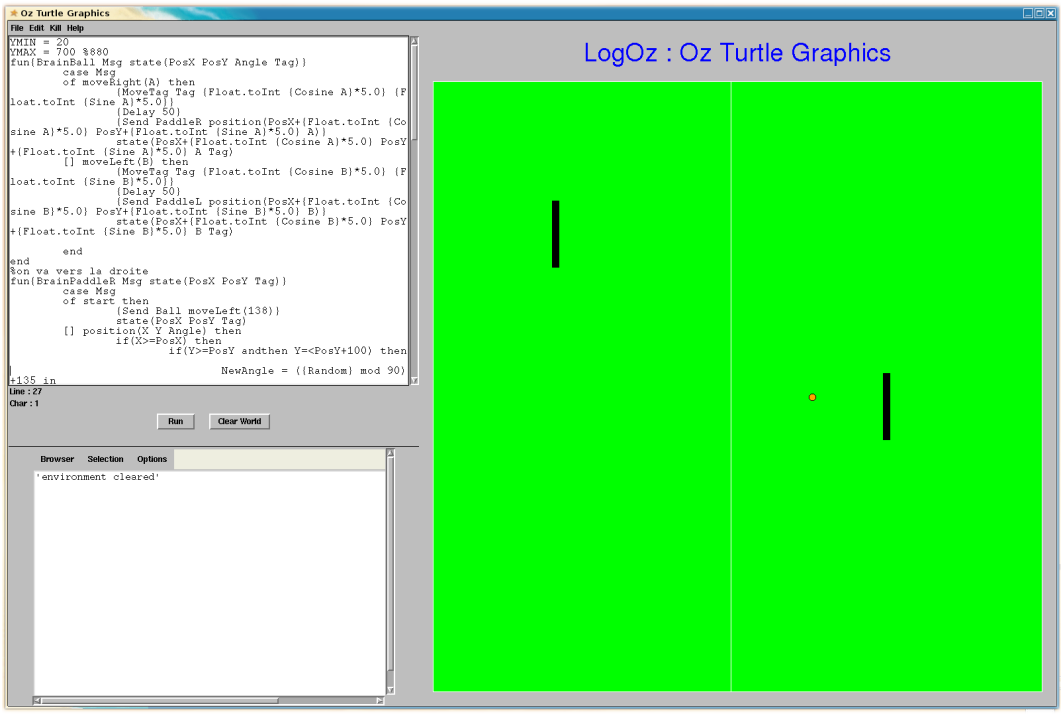

FIG. 2.23 – Un jeu de ping pong

2.6.5 Conclusion sur le cinquième micromonde

Ce cinquième micromonde offre des possibilités pratiquement sans limites. N'importe quel type et n'importe quel nombre d'agents peuvent être créés et peuvent interagir. Seuls les outils graphiques sont limités mais ceci devra être complété par les mémorants qui vont nous succéder l'année prochaine.

2.7 Sixième micromonde

Dans le cadre de notre mémoire, nous nous sommes arrêtés au cinquième micromonde. Par la suite il sera possible de développer un sixième micromonde, et bien d'autres encore. Ce sixième micromonde sera bien entendu une évolution du cinquième. Il serait enrichi essentiellement par le concept de **composants**. Dans cette section, nous allons donner les grandes idées ainsi que le fil conducteur qui, nous l'espérons, permettra à nos successeurs de poursuivre le développement.

L'accent sera tout d'abord mis sur le concept de *composants*. La programmation orientée composants consiste à utiliser une approche modulaire au niveau de l'architecture d'un projet informatique, ce qui permet d'assurer au logiciel une meilleure lisibilité et une

meilleure maintenance[17]. Lorsque l'on parle de composants, il s'agit de simples fichiers, contenant généralement du code compilé. Un composant regroupe un certain nombre de fonctionnalités qui peuvent être appelées depuis un programme externe, ou d'un client.

Concrètement, le monde dans lequel vit la Turtle pourrait être un composant. Les composants peuvent être implémentés comme des classes. Un monde serait constitué d'un ensemble d'agents, chacun étant également une classe, ce qui ouvrirait de nombreuses nouvelles portes. Cette technique permettrait de structurer les applications en couches.

La figure 2.24 est une façon dont pourrait être implémenté l'agent Turtle, qui serait alors un composant.

```
class Turtle
   attr agent

   meth init(Name X Y Angle PenUp C)
      agent:={NewTurtle state(X Y Angle PenUp C)}
   end
   meth forward(N)
      {Send @agent forward(N)}
   end
   meth left(N)
      {Send @agent left(N)}
   end
   meth right(N)
      {Send @agent right(N)}
   end
   meth name(N)
      {Send @agent name(N)}
   end
end
Tortue = {New Turtle init('mat' 440 340 270 true black)}
{Tortue left(90)}
{Tortue forward(100)}
```

FIG. 2.24 – Une classe Turtle

Un monde sera un composant composé, c'est-à-dire qu'il contiendra différents autres composants, et qu'il pourra être sauvegardé et restauré à nouveau plus tard. Ces autres composants seront des composants simples, des agents se trouvant dans le monde, ou des composants composés d'autres mondes contenus dans le premier monde. Il peut donc y avoir des mondes (des composants) imbriqués, et il est ainsi possible de former différentes couches. Dans ce micromonde, un agent peut donc être lui-même composé d'agents internes.

Il serait également utilse d'avoir des agents qui seraient des éléments de l'interface graphique. Ces agents pourraient par exemple être des boutons, des labels, des canevas ou bien d'autres éléments graphiques.

Un bouton serait par exemple un agent qui, lorsque l'on clique dessus, recevrait un message afin qu'il effectue une certaine action. Tous les éléments de l'interface pourraient ainsi être des composants. Un agent bouton pourrait bien évidemment aussi envoyer des messages à d'autres agents. L'interface serait alors un ensemble d'agents qui interagissent entre eux à la demande (ou non) de l'utilisateur.

2.8 Septième micromonde

Une prochaine évolution pourrait être une évolution vers la tolérance aux pannes. Un agent serait chargé de contrôler sans arrêt s'il y a un problème. En cas de panne d'un agent, celui-ci mourrait et un message serait envoyé à l'agent responsable du contrôle des pannes. Ce dernier décidera alors de l'action à effectuer. Cette action dépendra bien évidemment de l'agent concerné et du problème rencontré.

Cette tolérance aux pannes permettra une certaine évolution vers une meilleure robustesse : le système pourrait devenir un système auto-surveillé. Cela permettrait une économie de temps et d'énergie que l'utilisateur pourrait consacrer à d'autres tâches.

Chapitre 3

Environnement de Programmation : LogOz

3.1 Architecture du logiciel

La figure 3.1 présente l'architecture logique de notre implémentation.

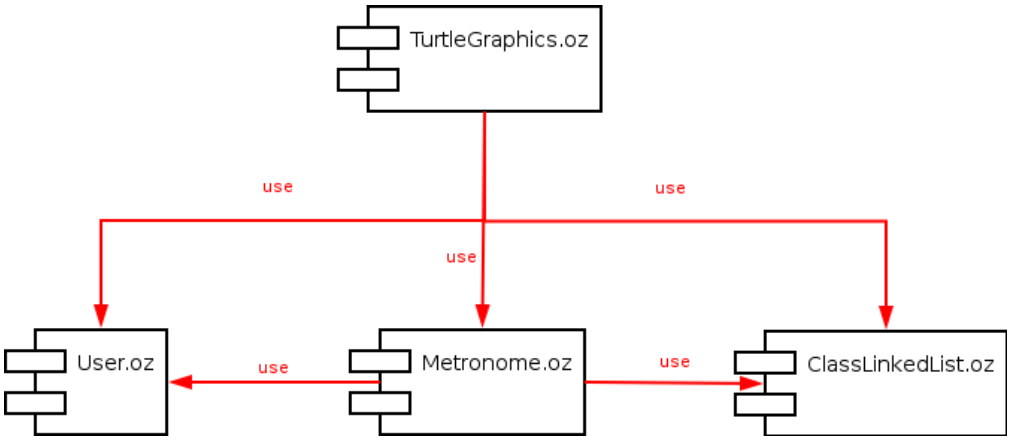

FIG. 3.1 – Architecture logique du logiciel

Nous avons divisé notre application en quatre parties, chacune correspondant à une unité de compilation indépendente des autres parties (functors). Chacun de ces foncteurs peut être vu comme un composant :

1. **TurtleGraphics.oz :** foncteur principal de l'application. Il contient toute l'interface graphique (les différents micromondes, le bloc notes, le Browser embarqué, les menus et les boutons). Il contient aussi tout ce qui est en rapport avec le compilateur et l'environnement, ainsi que tout ce qui touche aux agents de type *turtle*.
2. **Metronome.oz :** ce foncteur contient tout ce qui se rapporte aux agents Metronome, c'est-à-dire qu'il définit leurs fonctions de création et d'utilisation.

3. **User.oz :** ce foncteur contient les fonctions destinées à l'utilisateur, comme par exemple les fonctions mathématiques (cosinus, sinus, ...), les fonctions de création d'agents, etc. Ce foncteur définit également l'agent Browser (permettant l'affichage) ainsi que l'intégration du Browser à l'interface.
4. **ClassLinkedList.oz :** ce foncteur contient une classe qui implémente une liste chaînée. Cette liste chaînée est utilisée afin de garder une référence vers les agents créés par l'utilisateur.

3.2 Implémentation des agents

Les agents tels que les *Turtle*, les *Metronome* et les agents créés par l'utilisateur sont des port object qui utilisent la technique du message passing. Nous avons repris l'implémentation du port object telle qu'elle est décrite dans "Concepts, Techniques, and Models of Computer Programming"[15]. Nous l'avons quelque peu modifiée afin de permettre de tuer le thread à tout instant. La figure 3.2 montre l'implémentation du port object.

```
proc{NewPortObject Init Fun ?P ?Kill}
   proc{MsgLoop S State}
      case S of Msg|Sr then
         {MsgLoop Sr {Fun Msg State}}
      [] nil then skip
      end
   end
   Sin
in
   thread
      local TId={Thread.this} in
         proc{Kill}
            {Thread.terminate TId}
         end
      end
      {MsgLoop Sin Init}
   end
   {NewPort Sin P}
end
```

FIG. 3.2 – Implémentation du port object

La procédure *kill* permet de tuer un thread de manière brutale. Cette procédure est exécutée lorsque l'agent reçoit un message *kill*. De cette façon, l'agent s'autodétruit. Cette solution ne permet cependant pas d'arrêter un agent lorsque celui-ci effectue une boucle à l'infini. Ce message *kill* se mettrait dans la file des messages reçus par l'agent, et ne serait

traité qu'après tous les autres qui étaient là avant lui. Si l'utilisateur s'est trompé dans son programme et que l'agent est dans une boucle infinie, le message kill ainsi envoyé ne serait jamais traité. C'est pourquoi nous avons ajouté une option *kill all turtles* dans les menus qui sont présentés plus loin [1].

3.3 LogOz

Notre environnement de programmation a été écrit en Oz (version 1.3.1) [13], c'est pourquoi nous l'avons baptisé *LogOz*. Le programme est prévu pour pouvoir être exécuté sur une machine Linux. Il fonctionne également sur une machine Windows, bien que quelques défauts d'affichage persistent.

LogOz est un environnement permettant d'apprendre la programmation et ses concepts d'une manière simple et ludique. Notre environnement de programmation se base sur différents micromondes qui évoluent au fur et à mesure de l'état d'avancement de l'étudiant. La force de la méthode résulte dans le fait que dès le début l'étudiant utilise un langage de programmation réel, Oz. Nous avons développé une interface de programmation qui se doit d'être aussi conviviale et interactive que possible. L'interface, que nous avons appelée *Oz Turtle Graphics*, est donc une sorte d'IDE (Integrated Development Environment) pour les débutants en programmation. Elle est très simple à utiliser mais également très complète.

Nous avons choisi l'anglais comme langue pour notre environnement de programmation. L'anglais étant la langue la plus utilisée dans le domaine de l'informatique, ce choix nous a semblé assez évident. Le vocabulaire utilisé n'est néanmoins pas très compliqué, de sorte que tout étudiant ayant une connaissance au moins basique de l'anglais peut se servir du logiciel sans problème.

3.4 L'interface graphique

Nous avons utilisé QTk [16] pour réaliser l'interface graphique de notre environnement de programmation. Elle est composée de plusieurs grandes parties :

1. le choix du micromonde ;
2. le bloc notes ;
3. le browser ;
4. le monde de la tortue ;
5. les menus ;
6. les boutons.

La figure 3.3 donne une vue générale de notre environnement de programmation.

[1] Les menus sont présentés au point 3.4.5.

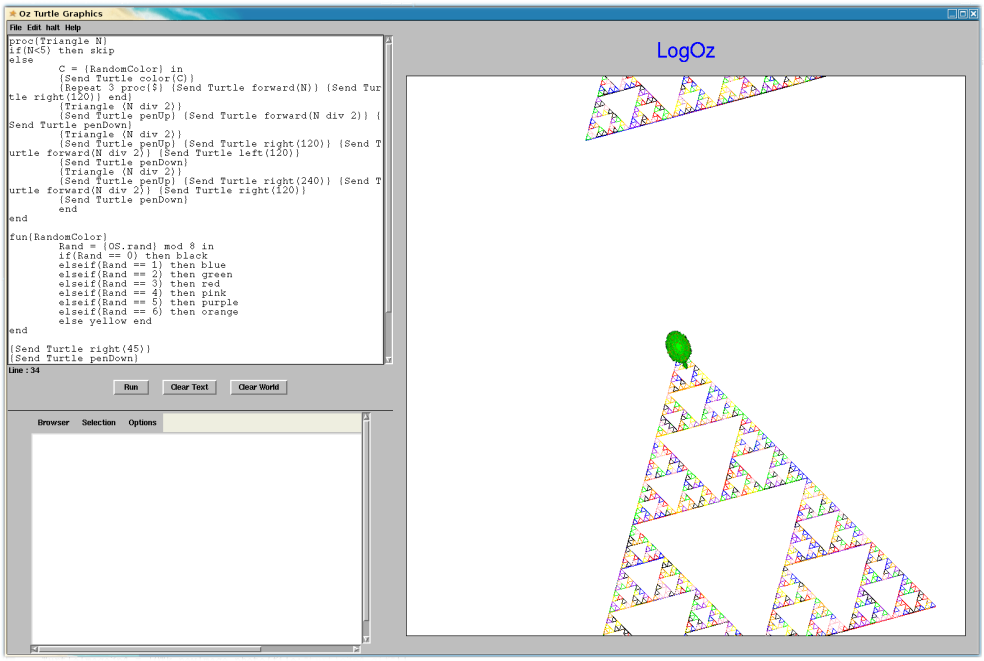

FIG. 3.3 – Oz Turtle Graphics

3.4.1 Le choix du micromonde

La figure 3.4 représente la fenêtre qui s'ouvre lorsque l'on lance le logiciel. Cette fenêtre permet de choisir le micromonde que l'on souhaite lancer. Chaque micromonde est en rapport avec certaines séances de cours.[2] Les détails sur les séances de cours se trouvent en annexe A. Une fois un micromonde sélectionné, l'environnement de programmation s'ouvre. Seules les fonctionnalités propres au micromonde choisi, ainsi que celles des micromondes précédents, sont présentes dans l'environnement afin de respecter l'approche par enrichissement de micromondes.

3.4.2 Le bloc notes ou notepad

La figure 3.5 représente le *bloc notes*. C'est dans ce bloc notes qu'est écrit le code qui sera compilé. Il prend l'apparence d'un simple fichier texte afin d'en simplifier l'utilisation. Il est possible d'écrire son code dans le bloc notes, de le sauvegarder et de charger le fichier par la suite. L'extenstion du fichier peut être choisie mais nous avons choisi l'extension *monFichier.oz* comme convention. Cela permet d'une part de sauvegarder un travail effectué précédemment, et d'autre part de programmer un code très grand dans un autre éditeur de textes plus pratique que le notepad (Emacs, gedit ...) -qui peut sembler fort petit pour certains codes- et de le charger dans le notepad par la suite afin de l'exécuter.

[2]Le lien entre les micromondes et les scéance de cours est expliqué au point *Séquencement des micromondes.*

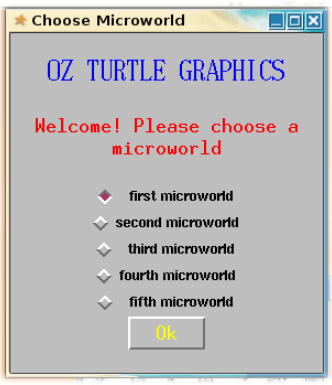

FIG. 3.4 – Choix du micromonde

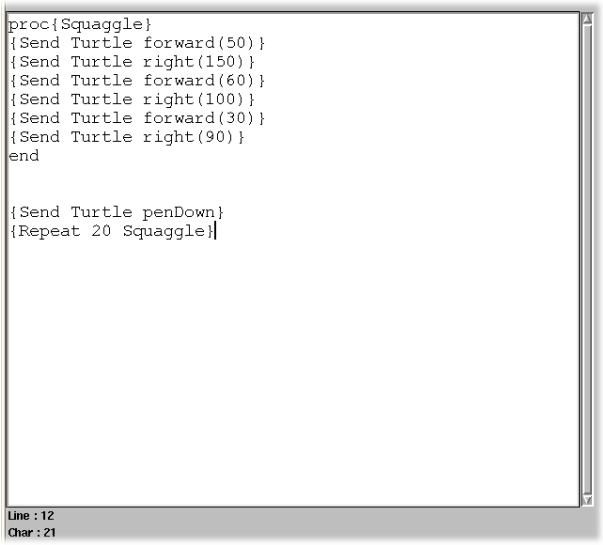

FIG. 3.5 – Bloc notes

Nous avons ajouté sous le bloc notes un label indiquant le numéro de la ligne et le numéro de la colonne auxquels le curseur se trouve. Cet ajout s'est avéré fort utile dans le cadre de la gestion des erreurs. En effet, le numéro de la ligne à laquelle l'erreur se produit est indiqué dans le browser.

Nous avions encore pensé à quelques améliorations possibles pour le bloc notes, comme par exemple y apporter une indentation ou de la coloration, car une fois que la taille d'un programme devient importante, il ne devient parfois plus très lisible.

3.4.3 Le browser

La figure 3.6 représente le *Browser*. Nous avons intégré le "Oz Browser" dans notre interface (embedded browser). Celui-ci peut être utilisé comme un simple browser (ex : {*Browse hello*}) ou en tant qu'agent Browser (ex : {*Send Browser hello*}). Il permet également l'affichage des erreurs de compilation. Lorsque le compilateur découvre une erreur, il l'indique dans le Browser. Le Browser permet donc à l'utilisateur d'afficher des données mais permet également au logiciel d'informer l'utilisateur.

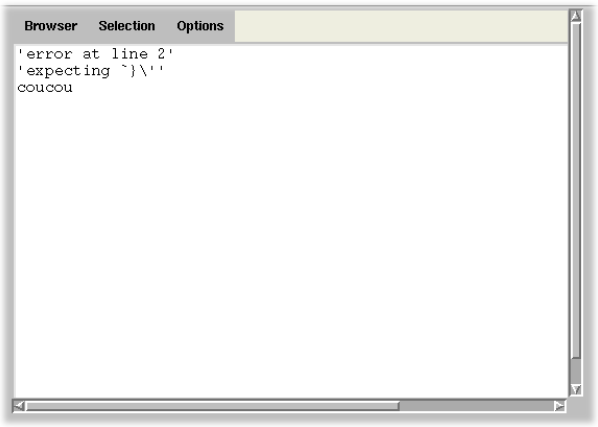

FIG. 3.6 – Browser

Le *Browser* sert également d'interface entre un agent et l'utilisateur. Par exemple, il permet à l'utilisateur de connaître certaines informations à propos de la tortue. Lorsque l'on clique sur la tortue, celle-ci affiche dans le browser son statut (position, penUp/penDown, couleur du trait, etc.).

Nous avons eu recours à une petite astuce pour pouvoir intégrer le Browser à l'interface graphique (Fig. 3.7). Il n'existe pas de "TK frame" en QTk. Nous avons donc créé un Browser Object similaire au browser prédéfini et nous l'avons placé dans un "label". Mais comme il nous était cependant impossible de modifier sa taille, nous avons alors ajouté un scrollframe afin qu'il ne prenne pas trop de place dans l'interface graphique.

Toutes les fonctionnalités du "Oz Browser" sont bien entendu conservées.

3.4.4 La feuille de dessin

La figure 3.8 présente le monde dans lequel la tortue évolue. Ce monde est représenté par une feuille de dessin. Le bouton "Clear world" permet de nettoyer la feuille dessin.

Plusieurs choix s'offraient à nous pour l'implémentation de la feuille de dessin. Un premier choix possible était une feuille de dessin finie. Lorsque la tortue arrive au bord, elle se retrouve bloquée, ce qui a pour inconvénient que le dessin à réaliser peut être

```
Bhandle
BrowseDesc = scrollframe(tdscrollbar:true lrscrollbar:true
                         label(handle:Bhandle width:1000 height:500))
BrowserObject = {New Browser.'class' init(origWindow:Bhandle)}
proc{Browse X}
   {BrowserObject browse(X)}
end
{BrowserObject createWindow}
{BrowserObject option(buffer size:100)}
```

FIG. 3.7 – Implémentation du browser embarqué

tronqué à cause de la taille finie de la surface disponible. L'espace de mouvement de la tortue est alors considérablement réduit.

Une seconde possibilité -et c'est pour celle-ci que nous avons opté- est de faire un tore. Lorsque la tortue franchit, par exemple, la limite supérieure de l'espace de dessin, elle se retrouve de l'autre côté, à la limite inférieure. L'avantage d'une telle solution est que le dessin n'est jamais tronqué, et que la tortue peut se déplacer dans n'importe quelle direction en parcourant de grandes distances.

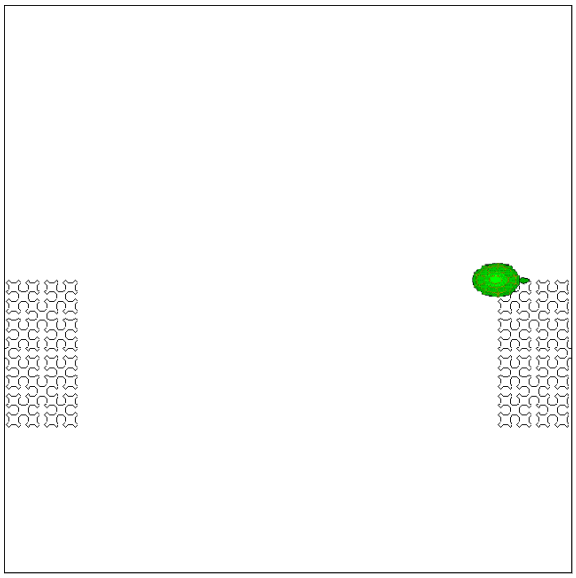

FIG. 3.8 – Le monde de la tortue : un tore

3.4.5 Les menus

La barre de menus est composée des quatre menus suivants :

1. File
2. Edit
3. Kill
4. Help

File

La figure 3.9 montre à quoi ressemble le menu *File*.

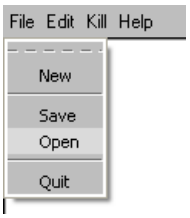

FIG. 3.9 – File menu

Ce menu comporte des fonctionnalités assez semblables à celles qui se trouvent dans beaucoup d'autres logiciels. Tous les éléments du menu sont disponibles dès le premier micromonde :

- *New* : l'environnement en mémoire ainsi que l'interface graphique sont remis à zéro. Tout ce qui avait été ajouté à l'environnement est retiré, le notepad et le browser sont vidés et le micromonde de la tortue est remis comme au départ, c'est-à-dire vide, contenant uniquement la tortue *Turtle* au centre du monde.
- *Save* : permet de sauver dans un fichier texte (.txt) le code qui se trouve dans le notepad. Ce code pourra ainsi être réutilisé plus tard (Fig. 3.10).

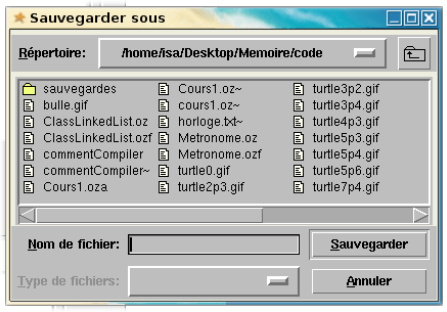

FIG. 3.10 – Sauvegarde d'un fichier

- *Open* : permet de charger un fichier dans le notepad (Fig. 3.11) ;
- *Quit* : quitte l'application (Fig. 3.12).

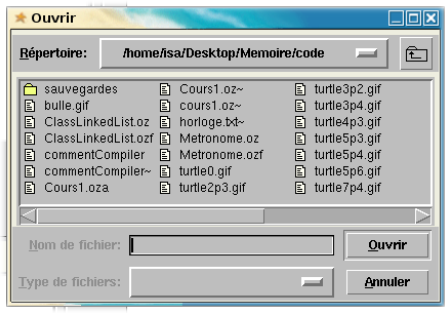

FIG. 3.11 – Ouverture d'un fichier

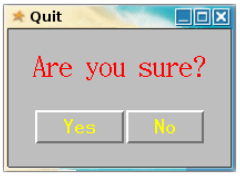

FIG. 3.12 – Quitter le programme

Edit

La figure 3.13 montre à quoi ressemble le menu *Edit*.

FIG. 3.13 – Menu Edit

Ce menu permet de modifier certains paramètres :

– *Delay* : cet élément du menu est disponible à partir du premier micromonde. Le "delay" est le temps d'attente (en millisecondes) de la tortue entre chacune des instructions qu'elle doit exécuter. Nous avons introduit un delay afin de permettre une meilleure visualisation des mouvements pour l'utilisateur. Ce delay peut être augmenté ou diminué jusqu'à 0ms.

– *Change World Interface* : cet élément du menu n'est disponible qu'à partir du cinquième micromonde. Dans ce micromonde nous avons introduit deux nouvelles interfaces de mondes, en plus de celle de la tortue : le cadran d'une horloge géante et un terrain de ping-pong, afin de permettre aux étudiants de réaliser eux-mêmes

de petites interfaces graphiques comme exercice. L'horloge géante est utilisée par les étudiants lorsqu'ils vont programmer les aiguilles d'une horloge. Ainsi les chiffres seront déjà inscrits à l'écran et il leur suffira d'ajouter les aiguilles qui bougent. Le terrain de ping-pong est utilisé lors du sixième cours. Le but est bien évidemment de créer un jeu de ping-pong. Le terrain sera donc en arrière-plan et il suffira aux étudiants d'ajouter les palettes ainsi que la balle pour pouvoir jouer. D'autres interfaces pourront encore être ajoutées dans le futur.

– *Clear Text* : cet élément du menu est disponible à partir du premier micromonde. Il permet d'effacer tout ce qu'il y a dans le bloc notes ; les données non sauvegardées seront perdues.

– *Clear Environment* : cet élément du menu est disponible à partir du premier micromonde. Il permet d'effacer de la mémoire toutes les variables déclarées par l'utilisateur, c'est-à-dire celles qui ont été ajoutées à l'environnement en mémoire. L'environnement initial du micromonde est donc rétabli. Si, par exemple, l'utilisateur avait déclaré une variable X, celle-ci n'existe plus après avoir cliqué sur ce bouton. L'environnement est donc remis à zéro.

Kill

La figure 3.14 montre à quoi ressemble le menu *Kill*.

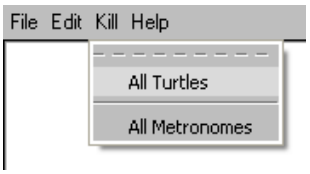

FIG. 3.14 – Menu Kill

Le menu *kill* permet d'arrêter les agents *Turtle* ou les agents *Metronome* de manière brutale. Il est composé de deux éléments :

– *allTurtles* : cet élément du menu est disponible à partir du second micromonde. L'arrêt de l'agent Turtle est parfois fort utile lorsque l'on se rend compte trop tard d'une erreur dans le code qui vient d'être lancé. Parfois cette exécution peut prendre longtemps, voire un temps infini dans le cas d'une boucle infinie, et il peut donc être utile de pouvoir l'arrêter avant d'en atteindre la fin.

– *allMetronomes* : cet élément du menu est disponible à partir du troisième micromonde. L'arrêt du Metronome peut également s'avérer utile lorsque plusieurs agents dépendent du message émis par le Metronome pour effectuer différentes actions. L'utilisateur peut mettre fin aux messages envoyés par le Metronome et donc aux actions qui s'ensuivent. Ce n'est qu'un raccourci de l'instruction {*Send MyClock stop*}.

Help

La figure 3.15 montre à quoi ressemble le menu *Help*.

FIG. 3.15 – Help Menu

Ce menu est composé de deux éléments :
– *Instructions* : contient une aide concernant les différentes commandes disponibles :

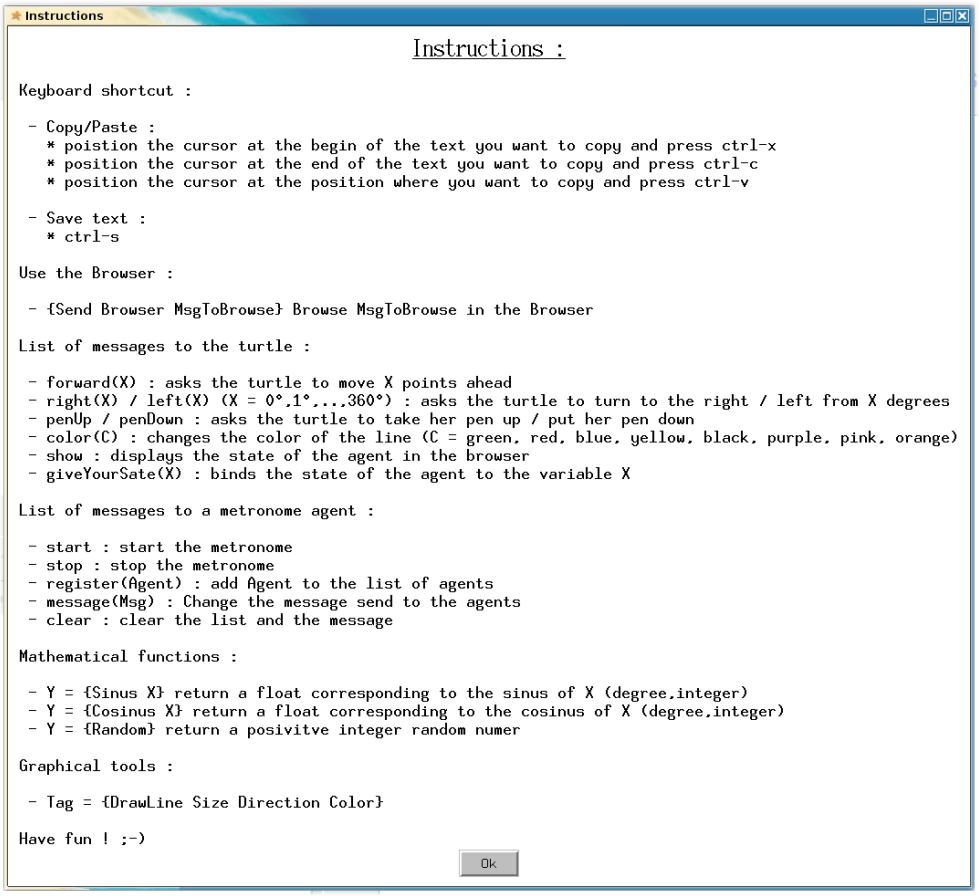

FIG. 3.16 – Menu d'aide

– *About* : contient les renseignements sur les auteurs et la version du logiciel :

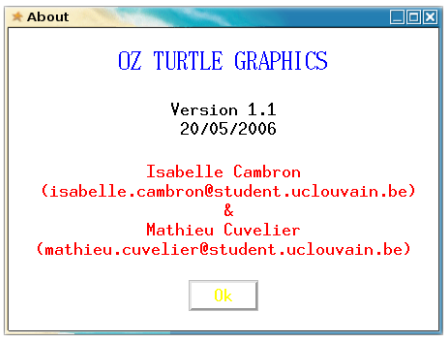

FIG. 3.17 – Informations sur le logiciel

3.4.6 Les boutons

La figure 3.18 présente les boutons *Run* et *Clear World* :

FIG. 3.18 – Boutons

- *Run* : compile et exécute le code qui se trouve dans le bloc notes. S'il y a une erreur à la compilation, celle-ci sera affichée dans la fenêtre d'affichage ;
- *Clear World* : réinitialise le monde actuel (Turtle World, Time World, ...), tout ce qui se trouvait dans le monde de l'interface sera effacé.

3.4.7 Le compilateur et l'environnement

Nous avons ajouté plusieurs outils dans l'environnement du compilateur de manière à simplifier au maximum la syntaxe du langage et permettre à l'utilisateur de progresser rapidement. Ces outils se comportent comme des mots clés du langage. Il ne faut pas les déclarer pour les utiliser. Ils se rajoutent à la liste des mots clés standards de Oz tels que Send, fun, proc, ...

Les outils ajoutés à l'environnement du compilateur

Le tableau 3.1 montre tous les outils rajoutés à l'environnement du compilateur.

Turtle	agent tortue existant dès le départ dans l'environnement De cette manière il n'est pas nécessaire de déclarer une tortue pour commencer à utiliser le logiciel
Browser	agent Browser qui permet d'afficher des informations dans la fenêtre d'affichage
Browse	browser classique de Oz que nous avons redirigé vers le Browser embarqué.
Repeat	procédure permettant de répéter N fois une procédure P
NewTurtle	fonction permettant de créer un nouvel agent tortue dans le monde
NewAgent	fonction permettant de créer un nouvel agent
NewMetronome	fonction permettant de créer un nouvel agent Metronome
Random	fonction qui renvoie un nombre aléatoire
Sine	fonction qui renvoie le sinus d'un angle en degrés
Cosine	fonction qui renvoie le cosinus d'un angle en degrés
Tangent	fonction qui renvoie la tangente d'un angle en degrés
DrawLine	fonction qui permet de dessiner une ligne dans le monde
DrawPaddle	fonction qui permet de dessiner une palette dans le monde
DeleteTag	procédure qui permet d'effacer un Tag du monde
MoveTag	procédure qui permet de déplacer un Tag dans le monde
DrawBall	fonction qui permet de dessiner une balle dans le monde

TAB. 3.1 – Outils rajoutés à l'environnement

Remarque : Le mot-clé *declare* est ajouté lors de la compilation de manière à ne pas devoir l'écrire à chaque fois afin de simplifier la prise en main du langage au début. Dès que l'on pousse sur le bouton run, ce mot clé est ajouté de la manière suivante :

```
{E enqueue(feedVirtualString("declare "#{TextHandle get($)}))}
```

Pour les micromondes qui seront développés dans le futur, il faudra peut-être enlever cet ajout. Par exemple, pour le multi-composants, l'étudiant sera certainement amené à implémenter plusieurs foncteurs. Dans ce cas, ce mot-clé doit être retiré. Il suffira alors de ne pas l'ajouter à l'environnement au moment de la compilation.

Chapitre 4

Séances de cours

Dans ce chapitre nous reprenons chaque séance de cours en pointant pour chacune ce que nous avons décidé d'enseigner aux étudiants. Ce chapitre prend en compte les améliorations que nous avons faites suite aux difficultés rencontrées lors des six séances de cours que nous avons données, et aux remarques des étudiants testeurs.

Nous n'avons pas séparé les séances de cours et les séances d'exercices. Les deux sont mélangées dans un but bien précis : nous pensons qu'en demandant aux étudiants de réaliser des exercices au fur et à mesure qu'ils apprennent de nouveaux concepts, ils réussiront mieux à les comprendre et à les utiliser par la suite. On trouve à la fin de chaque séance un ou plusieurs exercices récapitulatifs.

4.1 Séance 1 : Premiers pas - Micromonde 1

Dans ce premier cours nous allons introduire une série de concepts de base de la programmation et décrire en détail l'interface utilisateur et comment l'utiliser. Mais avant cela, nous définissons ce qu'est un agent et précisons son utilisé. Cette définition est la toute première du cours car nos micromondes sont composés d'agents. Ce concept est donc central et doit être bien compris par tous.

4.1.1 Les agents

Après avoir introduit ce concept, nous expliquons que certains agents prédéfinis existent déjà dans le micromonde. Il s'agit de deux agents distincts pour le premier micromonde : l'agent Browser et l'agent Turtle.
La communication avec ces agents se fait par l'envoi de messages du type

```
{Send Agent Message}
```

– L'agent Browser est un agent qui permet d'afficher dans la fenêtre d'affichage (le Browser) ce qui lui est demandé. Par exemple,

```
{Send Browser coucou}
```

affichera dans le Browser "coucou".

– L'agent Turtle est la tortue visible au milieu du monde. Toute une série de commandes peuvent lui être envoyées grâce au message *Send*. Le tableau récapitulatif contenant l'ensemble de ces commandes que nous fournissons aux étudiants et dont ils se serviront tout au long des travaux pratiques est le tableau 2.1 plus haut.
Un agent Turtle peut donc se déplacer dans toutes les directions dans son monde.

4.1.2 L'interface

Nous passons ensuite à une explication un peu plus détaillée de l'interface utilisateur. Un des objectifs de la première séance est d'apprendre aux étudiants à se servir de l'interface que nous avons développée pour eux. Jusqu'à présent ils connaissent les messages à envoyer à l'agent Turtle pour qu'il se déplace dans son monde, mais ils ne savent pas comment envoyer ces messages. Cette section décortique donc les grandes parties de l'interface graphique et explique l'utilité de chacune.

Les étudiants sont ensuite invités à faire quelques essais avec cette interface : envoyer des messages au Browser, déplacer la tortue dans son monde et lui faire faire quelques dessins simples, etc.

Dans cette partie le concept de *variable* et d'*atome* sont introduits. Ils permettent d'expliquer une partie de la syntaxe utilisée (cette syntaxe étant celle du Oz). Nous définissons également ce qu'est une instruction. Cela peut paraître inutile, mais il ne faut pas oublier que ce cours s'adresse à un public n'ayant encore jamais fait de programmation auparavant.

4.2 Séance 2 : Procédures - Micromonde 2

Dans cette section nous allons, comme le titre l'indique, expliquer ce qu'est une procédure et à quoi elle sert. Nous commençons par expliquer ce qu'est une procédure en général. Ensuite nous y ajoutons les arguments. Nous illustrons ensuite ce concept en montrant quelques exemples, et après cela nous passons aux dessins procéduraux. Tout au long de cette séance l'étudiant aura l'occasion de tester lui-même les exemples que nous lui montrons dans le cours. Tout à la fin, un exercice récapitulatif lui est proposé.

4.2.1 Les procédures

Cette première partie de cours s'attarde sur la définition précise d'une procédure et son mode d'emploi. Le concept de procédure est immédiatement illustré par un exemple, ce qui permet, dans ce cas-ci, de comprendre plus facilement de quoi il s'agit. La procédure de la figure 4.1 montre comment dessiner un carré dont le code pourra être réutilisé par la suite.

L'étudiant aura eu lors du premier cours l'occasion de dessiner un carré en écrivant l'ensemble des instructions nécessaires à sa réalisation. Ici, il découvre qu'il est possible de réutiliser plusieurs fois ce même code pour dessiner différents carrés.

```
proc{Carre}
   {Send Turtle penDown}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
end

{Carre}
```

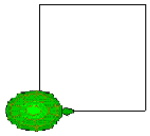

FIG. 4.1 – Carré de côté 100

4.2.2 Les arguments

Les arguments sont ensuite ajoutés aux procédures. L'étudiant découvre comment passer ces arguments à la procédure en les écrivant après le nom de cette dernière. Nous reprenons alors l'exemple du carré de la figure 4.1. Celui-ci va pouvoir prendre différentes tailles grâce à un argument passé à la procédure qui dessine le carré (Fig. 4.2).

```
proc{Carre X}
   {Send Turtle penDown}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
end

{Carre 200}
{Carre 100}
{Carre 50}
```

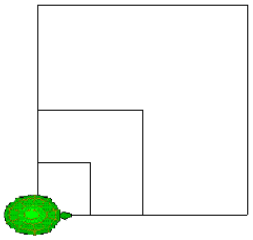

FIG. 4.2 – Carré avec argument

Cette fois l'étudiant découvre que son code peut être réutilisé afin de dessiner différents carrés, à différents endroits, et surtout avec des paramètres différents.

Une procédure qui sera fort utilisée est ensuite introduite : la procédure **Repeat**. Cette

procédure s'écrit de façon suivante :

$$\{Repeat\ N\ P\}$$

L'argument N représente le nombre de fois que la procédure P devra être exécutée. L'étudiant découvre alors une des caractéristiques les plus importantes de la programmation fonctionnelle : les procédures peuvent prendre d'autres procédures en argument.

La procédure Repeat va permettre de réaliser des dessins bien plus compliqués que ce qu'il était possible de faire avant, car il s'agit d'un énorme raccourci dans certains cas. Prenons pour exemple celui du code et du dessin d'un cercle. Cet exemple est présenté à la figure 2.5. Si la procédure Repeat n'existait pas, il faudrait appeler soi-même 360 fois la procédure Cercle, ce qui ne serait vraiment pas pratique. Le Repeat introduit donc un raccourci pour programmer.

4.2.3 Dessin procédural

Dans cette section une série de dessins réalisés grâce aux procédures sont présentés. La figure 4.3 présente l'exemple d'un oeil dessiné uniquement à l'aide de carrés.

Cet exemple reprend une grande partie des concepts vus jusqu'à présent. On peut en effet voir qu'une procédure en appelle une autre et que la procédure Repeat est plusieurs fois utilisée.

D'autres dessins tels qu'une fleur ou un tore sont également présentés.

Pour terminer cette section, quelques exercices sont proposés à l'étudiant.

4.3 Séance 3 : Opérateurs, conditionnelle et récursion - Micromonde 2

Pour commencer cette section, un petit rappel est fait sur les opérateurs arithmétiques et les opérateurs binaires. Ensuite nous abordons différents concepts : la conditionnelle avec la commande *if*, le pattern matching avec la commande *case of* et puis la récursion.

4.3.1 Opérateurs arithmétiques et binaires

A l'occasion de ce rappel, la différence entre les nombres entiers (Integer) et les nombres réels (Float) est expliquée. Cette différence est importante car dans certains exercices qui suivent des calculs en nombre à virgule seront nécessaires. De plus, il existe des opérateurs différents en fonction du type de nombre que l'on utilise (par exemple "/" pour la division entière et "*div*" pour la division de nombres réels).

Un rappel est également fait sur les opérateurs binaires. Ceux-ci sont semblables aux opérateurs binaires habituels. Il faut juste bien retenir la syntaxe qui peut différer d'un langage à un autre.

```
proc{Cercle}
   {Send Turtle color(black)}
   {Send Turtle forward(40)}
   {Send Turtle color(blue)}
   {Send Turtle forward(40)}
   {Send Turtle right(90)}
   {Repeat 2 proc{$}
      {Send Turtle color(black)}
      {Send Turtle forward(80)}
      {Send Turtle right(90)}
   end}
   {Send Turtle color(blue)}
   {Send Turtle forward(40)}
   {Send Turtle color(black)}
   {Send Turtle forward(40)}
   {Send Turtle right(90)}
end

proc{Oeil}
   {Send Turtle penDown}
   {Repeat 180 proc{$}
      {Cercle}
      {Send Turtle right(2)}
   end}
end

{Oeil}
```

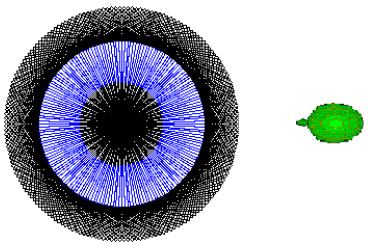

FIG. 4.3 – Oeil bleu

4.3.2 Conditionnelle : la commande *if*

Dans cette partie une définition de la commande *if* est donnée. Nous montrons également comment elle est utilisée.

```
if <x1> then
   <s1>
elseif <x2> then
   <s2>
else
   <s3>
end
```

Si la condition $x1$ est remplie, $s1$ est exécuté. Sinon, si la condition $s2$ est remplie, $s2$ est exécuté. Sinon, $s3$ sera exécuté.

Nous enchaînons par quelques exemples afin d'illustrer cette définition qui semble, au premier abord, assez compliquée.

Nous profitons de cette partie de cours pour expliquer comment définir une variable à l'intérieur d'une procédure. Il s'agira d'une variable locale. Cela illustre le concept de *portée d'une variable*.

4.3.3 Pattern matching : la commande *case of*

Comme pour la commande *if*, nous définissons tout d'abord ce qu'est la commande *case of* et à quoi elle sert. Nous montrons de quelle manière elle est utilisée.

```
case <x>
   of <l1> then <s1>
   [] <l2> then <s2>
   [] <l3> then <s3>
   else <s4>
end
```

On compare x d'abord à $l1$. Si les deux éléments correspondent, $s1$ sera exécuté. Sinon, on passe à la prochaine ligne. x sera alors comparé à $l2$ et ainsi de suite.
Dans tous les autres cas, c'est-à-dire si aucune correspondance n'est trouvée, $s4$ sera exécuté.

Cette explication est suivie de quelques exemples afin d'illustrer l'utilisation du *case of*.

4.3.4 La récursion

Cette section aborde un des concepts clés de la programmation : la récursion. Celle-ci est fort utilisée et très pratique. Nous commençons par donner une définition suivie d'un exemple qui l'illustre.

Définition

On dit qu'une procédure est récursive si elle s'appelle elle-même. La récursion permet de répéter plusieurs fois une action. Dans la récursion on distingue souvent 2 cas :

1. le cas de base, dans lequel la procédure se termine ;
2. le cas récursif, dans lequel se trouve l'appel récursif afin de boucler.
   Attention ! Toutes les instructions écrites après l'appel récursif seront sauvegardées sur la pile (stack) pour pouvoir être exécutées par la suite ! Cette notion de pile sera illustrée dans un exemple (Fig. 4.4) présenté plus tard.

Voici un exemple qui consiste à faire plusieurs $forward$ les uns après les autres en utilisant la récursion :

```
proc{MoveForward X}
   if(X == 0) then skip            % fin de la procédure
   else
      {Send Turtle forward(25)}    % avance de 25
      {Delay 1000}                 % attend pendant 1 sec
      {MoveForward X-1}            % appel récursif de la procédure
   end
end

{Send Turtle penDown}
{MoveForward 3}                    % appel de la procédure
```

Remarques :

1. X représente le nombre de fois que l'on souhaite exécuter un $forward$, c'est-à-dire le nombre d'appels récursifs qui seront faits ;
2. Le mot clé *skip* est utilisé pour quitter une procédure. Cette instruction permet donc de sortir de la procédure et de terminer le programme.

Nous expliquons en détail comment fonctionne ce programme simple, où se passe la récursion et quel effet elle aura.

Exemples

Nous enchaînons sur quelques exemples. Ces exemples ont pour but d'illustrer la récursion au travers de dessins qu'il est possible de réaliser dans le logiciel qui accompagne le cours.

Nous allons à la figure 4.4 reprendre un de ces dessins, celui d'un arbre récursif.

Une explication détaillée concernant ce qui se passe sur la pile lors de l'exécution de la procédure *Arbre* suit cet exemple, ainsi qu'une seconde approche légèrement différente.

Pour terminer ce cours, un exercice récapitulatif est proposé aux étudiants.

4.4 Séance 4 : Les agents - Micromonde 3

Nous démarrons ce cours par un rappel de la définition d'agents qui avait été vue lors du premier cours. Ensuite nous expliquons ce qu'est l'état d'un agent, et puis brièvement son cerveau, c'est-à-dire sa fonction de transition. Nous verrons par après comment créer soi-même un nouvel agent, et terminerons le cours en introduisant le concept de concurrence.

```
proc{Arbre L G}
   if(G==0) then skip
   else
      {Send Turtle forward(L)}
      {Send Turtle right(45)}
      {Arbre (L div 2) G-1}
      {Send Turtle left(90)}
      {Arbre (L div 2) G-1}
      {Send Turtle right(45)}
      {Send Turtle left(180)}
      {Send Turtle forward(L)}
      {Send Turtle left(180)}
   end
end

{Send Turtle right(90)}
{Send Turtle forward(100)}
{Send Turtle left(180)}
{Send Turtle penDown}
{Arbre 220 7}
```

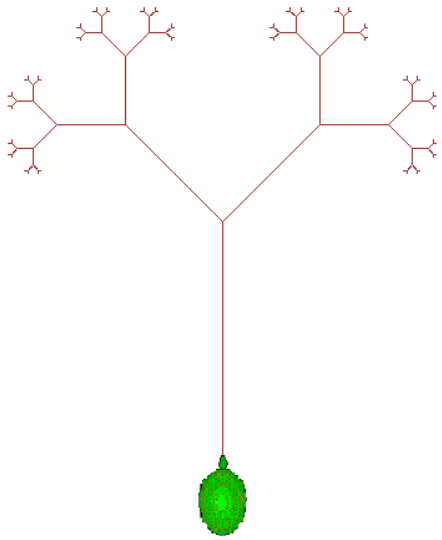

FIG. 4.4 – Arbre récursif

4.4.1 Rappel de la définition

En plus de la définition d'un agent donnée lors du cours 1, nous introduisons ici le fait qu'il puisse y avoir plusieurs agents. Ces agents peuvent communiquer entre eux et pour cela il faut une série de nouveaux mécanismes. L'état et la fonction de transition d'un agent vont être introduits comme nouveaux concepts que nous allons détailler dans les sections suivantes.

4.4.2 Etat

Chaque agent possède son propre état qui peut être composé d'éléments variés. L'état d'un agent correspond à l'état dans lequel il se trouve à un moment donné.

Nous allons dans cette section commencer par examiner l'état simplifié de l'agent Turtle. Nous ne présentons pas l'état complet car nous estimons que certains éléments ne sont d'aucune utilité pour l'étudiant. Nous avons donc retiré les éléments que nous n'avons pas jugés nécessaires pour la suite.

Afin de comprendre comment est fait cet état, il faut introduire le concept de *tuple*. En effet, l'état dans un port object est représenté sous forme d'un tuple.

Voici donc le tuple représentant l'état de l'agent Turtle :

```
state(Name pos(X Y) Direction PenDown Color)
```

L'état de la tortue est composé de 5 éléments distincts :

1. son nom : la tortue présente dans le monde s'appelle 'turtle'. Lorsque l'on crée une nouvelle tortue il faut toujours lui donner un nom. Ce nom est un atome ;
2. sa position : la position est composée de deux coordonnées, X et Y. Ces coordonnées permettent de situer la tortue dans l'espace à deux dimensions qu'est le monde dans lequel elle évolue. Les coordonnées X et Y sont mises à jour lors de chaque déplacement de la tortue, c'est-à-dire lors de chaque réception d'un message de type $forward(N)$;
3. sa direction : la direction permet de mémoriser vers où la tortue est actuellement orientée. A chaque envoi de message $left(N)$ ou $right(N)$ cette direction est mise à jour ;
4. PenDown : il s'agit d'un booléen qui indique si le crayon de la tortue est actuellement posé ou non. Le booléen passera à *vrai* lors de la réception d'un message du type *penDown*, et repassera à faux lors de la réception d'un message *penUp* ;
5. sa couleur : il s'agit tout simplement de la couleur dans laquelle les traits du crayon apparaissent à l'écran. Un choix de sept couleurs est possible : rouge (red), bleu (blue), vert (green), jaune (yellow), noir (black), mauve (purple) ou rose (pink).

L'état de la tortue peut être vu en cliquant une fois dessus à l'aide du bouton gauche de la souris.

4.4.3 Fonction de transition

Nous ne nous attardons que très peu sur la notion de fonction de transition dans ce cours. Elle sera vue en profondeur ultérieurement. Ce que nous disons tout de même c'est que la fonction de transition est comme le cerveau de l'agent. C'est dans cette fonction que se trouve l'ensemble des messages compris par l'agent. Lorsque l'agent reçoit un nouveau message, il vérifie si ce message se trouve déjà dans sa fonction de transition. Si oui, il effectue l'action qui avait été attachée à ce message. Sinon, l'agent effectue l'action par défaut (souvent il ne fait simplement rien). Le cerveau contient également l'état dans lequel l'agent se trouve. C'est grâce à la fonction de transition que l'état est mis à jour lors de chaque réception de message.

4.4.4 Création d'une nouvelle tortue

Dans le cours 5 nous aurons l'occasion de voir comment créer d'autres nouveaux agents, mais commençons tout d'abord par créer un nouvel agent Turtle.

Afin de pouvoir créer nous-mêmes une nouvelle tortue, il faut tout d'abord introduire le concept de *fonction*. En effet, nous allons utiliser la fonction *NewTurtle* afin de créer la nouvelle tortue. Nous commençons donc par définir ce qu'est une fonction, et exposer la différence entre une procédure et une fonction.

La fonction *NewTurtle* se présente sous la forme suivante :

$$X = \{NewTurtle\ State\}$$

La fonction renvoie un nouvel agent Turtle. Le nouvel agent est associé à l'identificateur X. Le cerveau de cet agent est identique à celui de la tortue d'origine. Il répond donc aux mêmes messages mais possède son propre état. Lorsque l'on crée une nouvelle tortue il faut lui passer comme argument son état initial *State* sous la forme suivante :

$$state('maTortue'\ 200\ 200\ 0\ false\ black)$$

Cet état représentera une tortue appelée 'maTortue' positionnée en (200,200) avec une direction de 0°, un crayon levé et une couleur de crayon noire. Une fois la nouvelle tortue créée, elle est immédiatement prête à recevoir des messages.

Voici un exemple de deux tortues dans le même monde :

```
MaTortue = {NewTurtle state('tortue' 200 200 0 true yellow)}
proc{Squaggle}
   {Send MaTortue forward(50)}
   {Send MaTortue right(150)}
   {Send MaTortue forward(60)}
   {Send MaTortue right(100)}
   {Send MaTortue forward(30)}
   {Send MaTortue right(90)}
end
{Repeat 20 Squaggle}

{Send Turtle penDown}
{Send Turtle forward(100)}
```

FIG. 4.5 – Deux tortues dans le même monde

4.4.5 La concurrence

Nous commençons tout d'abord par définir et expliquer ce qu'est la concurrence. Ensuite, pour illustrer ce nouveau concept, nous présentons un exemple de quatre tortues réalisant ensemble un dessin commun. Cet exemple est présenté à la figure 2.14 et 2.15.

Ce cours se termine à nouveau par quelques exercices récapitulatifs proposés aux étudiants.

4.5 Séance 5 : Le temps - Micromonde 4

L'objectif de cette séance est de mettre en évidence la concurrence ainsi que la communication entre différents agents. Cette séance s'appuie sur un autre type d'agent prédéfini : le *Metronome*.

4.5.1 L'agent Metronome

L'agent Metronome est prédéfini dans LogOz. Il peut envoyer exactement toutes les secondes un message à d'autres agents. Il n'est rien d'autre qu'un active object tout comme la tortue. Un tel agent a donc besoin d'une fonction de transition et d'un état. Tout comme pour la tortue, la fonction de transition est déjà existante dans le logiciel. Il suffit donc pour le créer de lui fournir son état. Voici la syntaxe utilisée afin de créer un Metronome :

$$MyClock = \{NewMetronome\ State\}$$

La fonction "NewMetronome" prend un seul argument : State. Le Metronome envoie un message à un certain nombre d'agents. Il s'attend donc à connaître à l'avance le message à envoyer et la liste d'agents auxquels il doit envoyer le message. L'état de notre nouvel agent est donc un tuple composé de deux variables :

1. la liste des agents à qui il doit envoyer le message ;
2. le message à envoyer à ces agents.

$$state([Agents]\ Msg)$$

Il faut introduire le concept de *liste* afin de permettre aux étudiants de l'utiliser pour créer la liste des agents auxquels le Metronome va envoyer son message.

Nous fournissons également un tableau contenant l'ensemble des messages compris par le Metronome2.2. Ce tableau sera fort utile aux étudiants par la suite.

Une fois créé, l'agent *Metronome* peut être lancé. Il revient donc aux étudiants eux-mêmes de lui envoyer un message afin qu'il démarre. Il faut également déterminer (avant ou après l'avoir lancé) la liste des agents à qui envoyer le message, ainsi que le message lui-même.

Afin d'illustrer ce principe, un exemple d'interaction entre plusieurs agents leur est présenté.

4.6 Séance 6 : Multi-agents - Micromonde 5

L'objectif de ce cours est d'apprendre à créer des agents autres que des agents Turtle et des agents Metronome. Ces agents auront un comportement propre, défini par le cerveau que son créateur lui attribuera. Nous nous focalisons dans ce cours sur la création d'un agent *horloge*, qui affichera l'heure sous forme d'une montre à aiguilles. Afin de créer cette horloge, nous utilisons l'agent Metronome, capable d'envoyer un message toutes les secondes à d'autres agents.

4.6.1 Création d'un nouvel agent

Jusqu'à présent nous avons créé des agents dont la fonction de transition, c'est-à-dire son cerveau, était intégrée au logiciel. En d'autres mots, nous ne pouvions pas définir le comportement des agents Turtle et Metronome. Dans cette section, nous nous intéressons à la création d'agents dont le comportement sera défini à l'aide d'une fonction de transition que nous allons définir nous-mêmes. Pour créer un agent il nous faut les éléments suivants :

1. une **fonction de transition** pour définir son comportement (cerveau)
2. un **état initial**.

Tout comme pour la création d'une tortue, il existe une fonction permettant de créer un nouvel agent :

$$MonAgent = \{NewAgent\ FonctionTransition\ Etat\}$$

La fonction *NewAgent* renvoie l'identifiant du nouvel agent créé et associe ce dernier à *MonAgent*. Cette fonction possède deux arguments : la fonction de transition et l'état initial. La fonction de transition de l'agent (son cerveau) doit être une fonction définie par l'utilisateur. L'état initial doit être l'état dans lequel l'agent se trouvera lors de sa création.

4.6.2 Fonction de transition

Une fonction de transition prend en règle générale deux arguments :

1. **un message :** ce message correspond au message reçu par l'agent. En fonction du message reçu, l'agent effectuera une action. Cette technique est communément appelée "message passing" (échange de message)[15]. L'action effectuée peut être, par exemple, l'envoi d'un message à un autre agent, un déplacement ou l'affichage d'une information. Il existe également une action par défaut dans le cas où l'agent reçoit un message qu'il ne comprend pas.
2. **un état :** il s'agit de l'état interne de l'agent. Celui-ci permet à l'agent d'avoir une mémoire. Cette mémoire peut l'aider à prendre certaines décisions ou tout simplement permet de mémoriser une série d'informations.

A la figure 4.6 se trouve l'exemple d'un agent qui se comporte comme un **compteur**. Son état initial sera 0.

Il est bien entendu possible d'utiliser l'agent Metronome pour envoyer des messages "tick" toutes les secondes à notre agent "Compteur". Dans ce cas, il faudrait remplacer les $\{Send\ MonAgent\ tick\}$ par l'appel au Metronome que nous avons déjà vu précédemment :

```
MyClock = {NewMetronome state([Compteur] tick)}
{Send MyClock start}
```

```
fun{Cerveau Msg State}
   case Msg
   of tick then
      {Send Browser State+1}
      State+1
   else
      {Send Browser 'I do not
       understand the message'#Msg}
      State
   end
end

Compteur = {NewAgent Cerveau 0}

{Send Compteur tick}
{Send Compteur tick}
{Send Compteur tack}
```

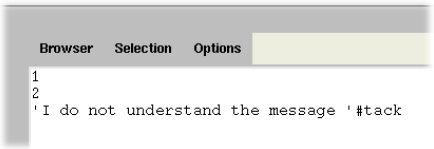

FIG. 4.6 – un compteur

4.6.3 Exercice : création d'une horloge

Pour clôturer ce cours nous avons prévu un exercice un peu plus conséquent, qui pourrait être vu comme un mini projet dans un cours réel. Il s'agit de créer différents agents afin de fabriquer une horloge. Certains éléments sont donnés, comme par exemple les fonctions de dessin des aiguilles, ou le monde de l'horloge (qui fait partie des différents mondes que l'on peut sélectionner dans le logiciel).

Pour réaliser l'horloge il faut utiliser l'agent Metronome qui enverra un message toutes les secondes à l'agent "Aiguille des secondes". Cet agent devra alors envoyer un ou des messages à l'agent "Aiguille des minutes", et ainsi de suite.

4.7 Projet : Programmation multi-agents - Micromonde 5

Ce projet illustre par un exemple très complet la façon dont des agents peuvent interagir. Il s'agit pour les étudiants de développer un jeu de ping-pong en suivant les directives et explications qui leur sont données. Un monde représentant un terrain de ping-pong est prévu pour cet exercice.

Il faudra créer différents agents :

- une balle ;
- une palette pour le joueur de gauche ;
- une palette pour le joueur de droite.

Différentes procédures et fonctions sont fournies aux étudiants pour leur faciliter la

tâche, surtout au niveau graphique. Il s'agit par exemple d'une procédure leur permettant de déplacer une image. Nous leur fournissons également l'état de chaque agent. L'étudiant aura pour objectif de faire fonctionner le jeu de ping-pong en créant les fonctions de transitions et en gérant l'affichage de chaque agent. Le résultat final se trouve à la figure 2.23.

D'autres jeux peuvent également être créés en utilisant le même principe. Par exemple le jeu des Space Invaders, le Game Of Life ou encore d'autres jeux comportant différents agents interagissant.

C'est ici que s'achèvent les cours que nous avons développés et que nous avons pu dispenser à quelques étudiants testeurs. Pour la suite, il s'agira uniquement d'un fil conducteur de ce qui pourrait se trouver dans les cours suivants. Cette vision n'est pas la seule possible et il existe bien évidemment de nombreuses autres façons de voir la suite de ce cours. Nous allons néanmoins exposer notre avis, en écrivant ce qui nous semblerait le plus logique et utile à enseigner par la suite.

Nous nous limitons à huit séances de cours, mais ce nombre pourrait être vu à la hausse en fonction des concepts qui seront ajoutés au fur et à mesure du développement.

Nous ne parlerons pas des évolutions nécessaires de l'environnement LogOz. Beaucoup de changements seront néanmoins nécessaires pour la suite du cours.

4.8 Séance 7 : les composants logiciels

Dans ce cours un nouveau concept d'agents logiciels pourrait être abordé, en commençant par les agents logiciels ordinaires, leur définition et l'utilisation possible. Ensuite il s'agira d'enchaîner avec des composants logiciels plus spécifiques qui pourraient être introduits dans l'interface graphique de l'environnement LogOz.

4.8.1 Introduction et définition

La prochaine étape du cours est de faire du monde dans lequel évoluent tous les agents, un agent. Il serait alors possible de créer un monde, d'interagir avec lui et puis de le sauvegarder, de l'interrompre et de le reprendre plus tard, de le démarrer et de l'arrêter. Tout ce qui est possible avec les agents créés dans les cours précédents serait possible avec le monde.

Il serait également possible de garder une trace de ce qui s'est passé dans le monde ainsi que de l'évolution de son état. Ceci permettrait de retourner à l'état précédent dans lequel était le monde, ou d'effacer certains événements.

Les agents Turtle et autres seraient alors des agents secondaires dans un grand agent principal. C'est ici que le concept de composants pourrait être introduit : le monde et l'ensemble des agents qu'il contient sont tous des composants. Il y a des composants primitifs : les agents situés dans le monde, et puis des composants composés : le monde, par exemple, qui est lui-même composé de différents composants (les agents). Un monde peut

contenir un autre monde. Les composants peuvent donc être imbriqués à l'infini.

Il s'agirait ici de bien définir le concept de composants pour les étudiants, afin qu'ils comprennent précisément de quoi il s'agit et à quoi cela sert.

Un composant logiciel est un élément du système ayant un comportement prédéfini et offrant un service prédéfini, capable de communiquer avec d'autres composants. Il existe différents critères à remplir pour compléter cette définition :

- réutilisabilité : un composant doit pouvoir être utilisé dans des contextes différents ;
- portabilités : non spécifique au contexte : un composant doit être indépendant du contexte dans lequel il se trouve. Il doit être portable ;
- composabilités : un composant doit pouvoir être composé d'autres composants.

Une définition plus simple pourrait être : un composant logiciel est un objet ayant une certaine spécification. Peu importe quelle est cette spécification, tant que l'objet a une spécification. C'est seulement en acceptant cette spécification qu'un objet devient un composant et obtient les caractéristiques telles que la réutilisabilité, etc.

4.8.2 Les composants logiciels dans l'interface utilisateur

Afin d'augmenter l'aspect orienté agent de l'environnement LogOz, il est possible de modifier certains éléments de l'interface graphique afin que ceux-ci deviennent également des agents, c'est-à-dire des composants logiciels. Ces agents apparaîtront donc comme des éléments de l'interface graphique : boutons, labels, canevas, etc.

Le système de gestion de ces éléments de l'interface graphique devra alors être complètement modifié. En effet, lorsque l'utilisateur cliquera par exemple sur un bouton, ce n'est plus une action liée à ce bouton qui devra être exécutée, mais c'est un message qui devra être envoyé à l'agent qui se cache derrière ce bouton. En fonction de ce message, l'agent effectuera alors une certaine action prédéfinie.

Il est possible ainsi d'ajouter toute une série d'éléments d'interface utilisateur en tant qu'agents qui peuvent interagir entre eux, et avec tous les autres agents se trouvant dans le monde. Il faudrait que les étudiants eux-mêmes puissent aussi en ajouter, en définissant pour chaque agent les actions à accomplir lors de la réception d'un message au travers de sa fonction de transition.

4.9 Séance 8 : composants logiciels concurrents

Les composants logiciels sont des composants concurrents, tout comme les agents le sont. En principe il n'y a pas d'interférences entre les différentes activités, à moins que le programmeur ne décide qu'il serait bon de faire coopérer certains composants entre eux afin de réaliser un objectif commun. Il peut également y avoir des problèmes de ressources partagées. Il faut se méfier de la concurrence et la gérer au mieux en utilisant des outils tels que les locks ou les moniteurs.

Le monde peut donc maintenant être composé de différents types d'agents. Nous avons déjà parlé des agents d'interface graphique et des agents tels que l'agent Turtle ou Metronome, mais il est également possible d'ajouter des agents de calculs, d'affichage, de sauvegarde, de mises à jour, et bien d'autres encore !

Le monde est donc un monde composé exclusivement d'agents, et donc de composants. Il est possible de choisir de faire évoluer ce cours dans de nombreuses directions, car les composants logiciels sont des concepts très riches.

4.10 Séance 9 : traîtement de pannes

L'on pourrait imaginer pour cette séance un système de gestion des pannes par un (ou plusieurs) agent(s) bien défini(s). Ce système permettrait une gestion assez particulière d'erreurs ou de problèmes : s'il y a un problème avec un agent, celui-ci meurt et un message est envoyé à l'agent chargé de gérer les pannes afin de lui signaler qu'il y a eu un problème, et qu'un agent est mort. L'agent gestionnaire des pannes devra donc réagir à ce message en fonction de ce que contenait le message, et aussi en fonction de l'agent qui a eu le problème. Il est également possible qu'un problème affecte plusieurs agents. Dans ce cas, le gestionnaire de pannes devra pouvoir gérer l'ensemble des messages qui lui sont envoyés simultanément.

Prenons par exemple deux agents qui s'occupent ensemble de l'affichage de données pour l'utilisateur. Les deux agents ne peuvent rien faire l'un sans l'autre. Si un problème survient pour un des deux agents, l'agent gestionnaire des pannes devra immédiatement être mis au courant. Si un des deux agents meurt, il faut rapidement prendre une décision en ce qui concerne le rôle que tenait cet agent. Il faut au plus vite le remplacer ou alors le recréer, sinon le second agent pourrait à son tour tomber en panne.

Voilà de quoi l'agent gestionnaire des pannes devra s'occuper : s'assurer que le système tourne toujours, et trouver des solutions lorsque des agents tombent en panne. Il pourrait avoir différentes réactions en fonction du problème rencontré et en fonction de l'agent qui est mort.

Chapitre 5

Evaluation

Ce chapitre a pour objectif de mettre en évidence d'une part la méthodologie que nous avons appliquée afin de donner les cours que nous avons développés, et d'autre part le feedback que nous avons obtenu de nos étudiants testeurs concernant de la méthode que nous avons élaborée.

Dans la première section nous expliquons comment nous nous y sommes pris pour recruter des étudiants et leur donner les séances de cours. Nous expliquons ensuite rapidement quel était le profil des différents étudiants testeurs. Nous parlerons aussi des difficultés que nous avons rencontrées tout au long des heures de cours, tant au niveau de l'implémentation de notre interface qu'au niveau de la matière enseignée. Nous développons ensuite les points positifs qui ont été relevés par nos étudiants testeurs, ainsi que ceux que nous avons constaté par nous-mêmes. Pour finir, nous évoquons les changements à apporter ou que nous avons déjà apportés à la méthodologie ainsi qu'à notre logiciel.

5.1 Méthodologie

Pour tester la méthode d'enseignement de la programmation par l'enrichissement progressif des micromondes multiagents il nous fallait des étudiants motivés, prêts à suivre quelques heures de cours et d'exercices. Idéalement ces étudiants devaient être en rhétorique ou en première année à l'université.

Nous avons tout d'abord cherché des volontaires du côté des élèves de l'enseignement secondaire. Mais n'ayant reçu que trop peu de réponses nous avons également cherché des étudiants volontaires en première année universitaire. Nous avons finalement trouvé une dizaine de personnes disposées à suivre notre formation.

Pour donner ces cours, il nous fallait également une salle informatique afin de permettre aux étudiants testeurs de réaliser les exercices à l'aide du logiciel que nous avons développé. Nous nous sommes donc arrangés avec la faculté INGI qui a mis à notre disposition la salle SINF ainsi que tout le matériel nécessaire. Les cours ont finalement eu lieu durant trois soirées, la première semaine après les vacances de Pâques. Chacune de ces séances a

duré à peu près trois heures.

5.2 Profils des étudiants

Les premiers contacts se sont très bien déroulés. Les étudiants qui se sont présentés étaient tous motivés pour apprendre. Nous avions réussi à rassembler une petite dizaine de personnes d'horizons et de formations fort différentes (rhétoriciens, étudiants en droit, histoire, économie, etc.). Leurs connaissances préalables concernant la programmation et l'informatique en général étaient aussi très différentes. Certains ont donc mis un peu plus de temps que les autres à se familiariser avec notre interface, mais tous ont rapidement réussi à la maîtriser. C'était là déjà un départ encourageant !

5.3 Problèmes rencontrés et tentatives de solutions

Nous avons tout au long de nos cours rencontré une série de petits problèmes que nous pourrions qualifier de mineurs. En effet, il n'y a pas eu de grand problème que ce soit au niveau de l'interface et du logiciel, ou concernant le cours lui-même. Ce point nous semble très important à relever car il témoigne d'une bonne préparation du terrain. Le fait que nous soyons nous-mêmes encore étudiants a probablement joué un rôle important à ce niveau car nous savons quelles sont les lacunes des méthodes d'enseignement actuelles et quels sont les points qui ont été plus difficiles à comprendre pour nous.

Nous ne pouvons néanmoins pas prétendre que l'ensemble des cours était parfait. Loin de là. Voici donc quelques problèmes auxquels nous avons dû faire face durant les heures de cours.

Le premier problème rencontré se situait au niveau de l'interface graphique. En effet, un bouton "Clear Text" qui servait à effacer l'ensemble du texte écrit dans le notepad, se trouvait entre deux autres boutons fort utilisés : le bouton "Run" qui sert à exécuter le code, et le bouton "Clear World" qui est destiné à effacer tout ce qui se trouve dans le monde de la tortue. Ce bouton ne se trouvait donc pas vraiment à un endroit stratégique : beaucoup d'étudiants se sont trompés en voulant cliquer sur un des boutons voisins et ont ainsi effacé tout leur code ! Erreur assez frustrante. Il nous a donc été demandé de retirer ce bouton et de le mettre à un endroit moins accessible : dans les menus déroulants.

Un second problème concernait la syntaxe du Oz. Nous n'avions pas bien défini certains aspects tels que "où et quand mettre des majuscules ? des minuscules ?" ou encore "quand mettre des parenthèses ou des accolades ?", "à quoi servent les '%' ?". Nous avons pour cela ajouté certains commentaires dans les notes de cours.

Un autre problème a ensuite été soulevé concernant les arguments des procédures. Le nombre d'arguments que l'on peut passer à une procédure et comment mettre plusieurs arguments ne semblait pas clair. Nous avons alors également ajouté quelques explications

à cet effet.

La procédure {*Repeat N P*} a, en général, été bien comprise par tous les étudiants. Seul point d'ombre : est-il possible de passer une procédure définie précédemment dans la procédure P ? Nous avons donc décidé d'illustrer cela par un exemple qui se trouve juste après l'explication du Repeat dans le cours.

Un problème -si nous pouvons réellement qualifier de 'poblème'- plus général que nos étudiants ont relevé est le fait qu'une variable n'est assignable qu'une seule fois. Apparemment ils considéraient cet aspect de la programmation comme illogique. Il est vrai que dans d'autres langages qui ne sont pas fonctionnels il est possible de faire, par exemple, X = X + 1, alors qu'en Oz -qui est un langage fonctionnel- cela n'est pas possible. Il s'agit tout simplement d'un aspect des langages fonctionnels dont il faut essayer d'expliquer au mieux le fonctionnement et la logique.

Lors du cours 3, le concept de récursion était rapidement clair pour tout le monde. Néanmoins, lors de l'exemple du dessin de l'arbre de façon récursive, la plupart des étudiants avaient des difficultés à voir dans quel ordre les instructions étaient exécutées. Nous leur avons alors longuement expliqué le concept de pile (stack) et la façon dont les instructions étaient placées dessus et ensuite récupérées. Cette façon de voir les choses n'était malgré cela pas évidente à comprendre pour tout le monde. Nous nous sommes alors demandés s'il était bien à sa place dans ce troisième cours. Nous avons tout de même fini par conclure que oui, car il est nécessaire afin que les étudiants puissent comprendre comment une procédure récursive est exécutée par la machine, et dans quel ordre les instructions seront récupérées.

Dans le cours 4 nous avons introduit la notion de *fonction* afin de créer de nouvelles fonctions de transition pour les agents. Quelques difficultés ont alors été éprouvées par les étudiants lorsqu'il s'agissait de comprendre la différence principale entre une fonction et une procédure : la fonction renvoie toujours une valeur alors que la procédure non. En Oz il n'y a pas de mot clé "return" pour renvoyer une valeur. Nous avons donc essayé de donner plus d'exemples dans le cours, et d'expliquer d'une manière un peu différente ce que fait une fonction et comment celle-ci se comporte.

Lorsque dans le chapitre 5 nous avons proposé aux étudiants de créer les fonctions de transition, le problème de la valeur renvoyée par une fonction s'est à nouveau posé. En effet, la fonction de transition doit à chaque fois renvoyer le nouvel état de l'agent. Là encore nous avons modifié un peu les explications fournies dans le cours.

L'ensemble de ces modifications apportées à la suite des commentaires faits par nos étudiants volontaires devra bien évidemment encore être testé avec d'autres étudiants, afin d'en vérifier l'efficacité.

5.4 Points positifs

Nous avons eu plusieurs échos très positifs à la suite des cours que nous avons donnés. Tout d'abord nous avons été félicités pour la clarté des cours présentés. Nous avions préparé pour chaque cours des supports que nous avons fournis aux étudiants présents. En plus de cela nous projetions le cours ainsi que des exercices faits en direct devant eux. Les exposés étaient eux aussi clairs, et nous avions des réponses à toutes leurs questions. Certains avaient parfois un peu plus de mal à suivre mais comme nous étions deux à donner les cours, il était possible de donner quelques explications supplémentaires en particulier. Ceci a permis à tous de rester motivés pour suivre le cours jusqu'au bout

Un autre point qui a été fort apprécié est le fait que nous proposions des exercices tout au long des cours. Nous pensons que pour bien comprendre un concept, il est important de le mettre en pratique. Nous avions donc prévu des exercices qui avaient pour seule difficulté un concept bien précis que nous venions de leur apprendre, afin qu'ils se familiarisent avec celui-ci. A la fin de chaque cours un ou plusieurs exercices récapitulatifs étaient également proposés. De cette manière, les étudiants peuvent tout d'abord bien intégrer chaque concept séparément, et ensuite les mettre tous en relation. De plus, cette façon de faire permet une certaine interactivité des étudiants. Ils ne sont pas obligés de suivre des heures de cours théoriques avant d'enfin pouvoir commencer à mettre en application ce qu'ils ont appris.

L'interface a également reçu différents commentaires positifs : elle est claire, pratique et facile à utiliser. Il existait néanmoins un problème au niveau des boutons au bas du bloc notes, comme mentionné précédemment, mais ce problème a été facilement résolu. Les autres boutons et menus sont eux aisément accessibles et l'affichage est clair.

Il reste tout de même quelques problèmes d'affichage dans l'intégration du Browser à l'interface mais il s'agit là d'un problème qui pourrait faire l'objet d'une autre étude.

La méthode d'enseignement que nous avons utilisée a également été fort appréciée. Nous avons essayé d'expliquer tout ce que nous utilisions dans les programmes. Contrairement aux méthodes d'enseignement utilisées aujourd'hui en première année à l'université, les étudiants ont ici pu comprendre tout ce qui se cachait derrière des lignes de code qui semblaient parfois bien compliquées.

L'approche des micromondes était un réel avantage pour ce cours. Nous avons pu, comme il était prévu, augmenter la difficulté au fur et à mesure de notre avancement. Ce n'est qu'une fois que l'on était bien sûrs que tout le monde avait compris les concepts enseignés que nous passions à la prochaine couche. Cette technique nous a permis de nous assurer que tout le monde suivait bien et que chacun était prêt à passer à l'étape suivante.

5.5 Changements apportés au logiciel

En donnant les cours, nous avons constaté qu'il existait différents bogues mineurs dans notre logiciel. Nous avons également ajouté quelques fonctionnalités manquantes d'après nos étudiants testeurs.

Voici la liste des modifications apportées :
- Nous avons rajouté un message *show* pour les agents Metronome permettant de voir l'état de cet agent et afin de savoir si celui-ci est *on* ou *off* ;
- Un message *name* a également été ajouté pour les agents Turtle permettant de modifier le nom attribué à un agent ;
- Nous avons rajouté une fonctionnalité copier/coller (CTRL-C CTRL-V) dans le bloc notes permettant de copier des parties de code plus facilement ;
- Nous avons également rajouté un raccourci clavier pour sauver un code (CTRL-S) ;
- Un étudiant a remarqué une erreur lorsque l'on fermait la fenêtre du logiciel (quitter le logiciel sans utiliser le bouton "quit"). Cette erreur a été corrigée ;
- Nous avons également déplacé le bouton "clear text" suite à la demande des étudiants. Ce bouton se trouvait à un endroit trop accessible ;
- Plusieurs petites erreurs dans le code ont également été détectées et corrigées.

Conclusion

A l'issue de ce travail nous avons réussi à développer une nouvelle méthode d'apprentissage de la programmation. Cette méthode comprend une première série de séances de cours accompagnées d'un prototype du logiciel "Oz Turtle Graphics" totalement opérationnel. La méthode ainsi développée a été testée et à fourni des résultats encourageants. Le logiciel a lui aussi été utilisé par plusieurs personnes déjà. Les différents feedbacks obtenus nous ont permis d'améliorer ce qui avait déjà été fait et d'aboutir à la version que nous avons présentée dans ce document.

Six séances de cours ainsi que les exercices s'y rapportant se trouvent en annexe. Ces séances de cours ont été réparties en cinq micromondes successifs, chacun de ces micromondes étant accessible lors du lancement du logiciel. Les séances contiennent les concepts de base de la programmation mais aussi des concepts plus avancés tels que les procédures, la récursion et la concurrence.

Nous avons eu l'occasion au cours de l'année de tester la méthode développée sur une petite dizaine d'étudiants. Le feedback obtenu nous a montré que nous sommes sur la bonne voie. Le logiciel développé ainsi que les supports de cours et les exercices sont déjà un bon point de départ. Le séquencement des concepts et des micromondes est également au point. Le prototype "Oz Turtle Graphics" que nous avons développé est entièrement opérationnel. Cinq modes différents existent et sont proposés au lancement du logiciel. L'étudiant fait son choix en fonction du micromonde dans lequel il désire se trouver.

Dans les chapitres 2 et 4 nous présentons les idées ainsi qu' un fil conducteur de ce qui devrait et pourrait être fait par la suite afin d'améliorer, mais surtout d'étendre l'approche à des concepts bien plus nombreux.

Pour la suite de ce projet nous pensons qu'il serait utile de trouver, dès le début de l'année scolaire, un professeur d'école secondaire qui serait prêt à consacrer avec toute sa classe de rhétorique quelques heures de cours tout au long de l'année à cet apprentissage. Nous pensons qu'un plus grand groupe de personnes dont les membres ne sont pas tous spécialement intéressés par la programmation constituerait un bon groupe de test. Il sera alors possible de voir comment les étudiants accrochent, si d'une séance à l'autre ils se souviennent de ce qui a été fait précédemment, si ceux qui au départ n'aimaient pas du tout finissent par apprécier la programmation, et bien d'autres constatations encore. Etaler les

séances tout au long de l'année nous semble être le meilleur moyen pour tester l'efficacité de cette méthode d'apprentissage.

Il faudra également ajouter des micromondes à ceux que nous avons déjà créés. Actuellement il y en a cinq, l'objectif final serait d'en contenir au moins le double. Il reste encore beaucoup de concepts et techniques à enseigner à des étudiants de première année pour qu'ils arrivent en seconde année avec un bon bagage.

Dans le futur l'application accompagnant le cours pourrait être rendue disponible sur Internet à la manière d'un applet Java. Il ne serait alors plus requis de posséder sur son ordinateur les éléments nécessaires à l'exécution d'un programme Oz. Le logiciel serait ainsi beaucoup plus portable et disponible pour tous.

Nous avons eu la chance et l'opportunité de pouvoir développer ce logiciel dans le cadre d'une étude universitaire. Ce développement aurait été impossible dans une entreprise car non rentable dès le début. L'université peut ainsi participer à son rôle d'entreprise citoyenne, elle peut à travers ce genre de recherches anticiper les besoins des entreprises et des personnes.

Nous espérons avoir par le présent travail contribué à rendre accessible au plus grand nombre une méthode d'apprentissage efficiente de la programmation. Nous espérons que cette approche se répande dans d'autres endroits dans le monde, outre la Colombie où Andrès Sandoval Beccerra, avec qui nous avons été en relation, travaille également sur un projet du même genre. Certes il y a encore beaucoup de chemin à parcourir avant que la méthode ne soit tout à fait au point, mais pour l'année prochaine la relève est déjà assurée. Nous pouvons rêver qu'un jour cette méthode et le code source du logiciel l'accompagnant soient rendus disponibles sur Internet afin que tous ceux qui le désirent puissent l'enrichir.

Bibliographie

[1] Object-oriented programming concepts. *The Java Tutorial*. `java.sun.com/docs/books/tutorial/java/concepts/index.html`.

[2] Object-oriented programming. *Wikipédia, L'encyclopédie libre*. `www.wikipedia.org`.

[3] Programmation fonctionnelle. *Université de Mons-Hainaut*. `www.umh.ac.be/genlog`.

[4] Imperative programming. *Wikipédia, L'encyclopédie libre*. `www.wikipedia.org`.

[5] Définition du micromonde. `www.gpcservices.com/dictionnaire/M/micromonde.html`.

[6] Seymour Papert. *Mind-Storms Children, Computers and Powerful Ideas*. The Harverster Press, 1985.

[7] Boris Allan. *Le Logo*. Belin, 1984.

[8] M. Yazdani. *New Horizons in Educational Computing*. Ellis Horwood, 1987.

[9] Logo programming language. *Wikipédia, L'encyclopédie libre*. `www.wikipedia.org`.

[10] Squeak. `www.squeak.org/`.

[11] Smalltalk. `www.smalltalk.org`.

[12] Introduction aux agents et systèmes multi-agents. *Politechnica University of Bucharest - 2002*. `turing.cs.pub.ro/auf2/html/chapters/chapter1/`.

[13] Mozart. *The Mozart Programming System*. `www.mozart-oz.org`.

[14] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman & Company, 1983.

[15] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[16] Qtk. *Graphical User Interface Design for Oz*. `www.mozart-oz.org/documentation/mozart-stdlib/wp/qtk/html/`.

[17] La programmation orientée composants. *Wikipédia, L'encyclopédie libre*. `www.wikipedia.org`.

[18] Sigcse. *SIGCSE Bulletin 06*. Conference Proceeding.

[19] Cs101 project. `www.CS101.org`.

[20] Toontalk. `www.toontalk.com/`.

Annexe A

Cours

A.1 Cours 1 : Premiers pas - Micromonde 1

A.1.1 Introduction

Bonjour à tous ! Le cours que vous allez suivre aujourd'hui et dans les deux jours qui suivent est un cours expérimental pour des étudiants n'ayant jamais fait de programmation. Il s'agit d'un cours qui se base sur les concepts de la programmation, et non pas sur un langage en particulier comme c'est souvent le cas dans la première année d'études à l'université. Nous allons également essayer de vous expliquer **tout** ce que vous allez utiliser, sans laisser aucune zone d'ombre et sans vous donner de formules toutes faites que vous ne comprendrez que dans quelques années, et encore ...

Nous allons vous parler d'agents, de procédures, de variables, de fonctions, de récursion et de bien d'autres choses encore ! Notre objectif est que d'ici à la fin de nos cours vous puissiez placer un concept sur chacun de ces mots, mots qui vous sont peut-être encore inconnus aujourd'hui.

Comme mentionné précédemment, ce cours est expérimental. Cela veut donc dire qu'il est possible que certaines choses ne soient parfois pas expliquées suffisamment à votre goût. N'hésitez pas à nous demander des explications supplémentaires ! Nous espérons également recevoir un feedback sur ce que vous avez appris, ce que vous avez moins bien compris, ce qu'il faudrait changer, améliorer ou sur n'importe quelle autre chose dont vous voudriez nous faire part.

Un tout grand merci pour votre aide et let the show begin !

A.1.2 Les agents

Définition : Un agent est une entité avec laquelle il est possible de communiquer grâce à un certain nombre de messages bien définis qui peuvent être compris par cet agent.

Un agent peut donc recevoir des messages et agir en conséquent. Il peut également envoyer lui-même un message à un autre agent qui réagira. Il peut donc y avoir plusieurs agents qui communiquent entre-eux et qui interagissent.

Pour commencer nous allons nous contenter d'un seul agent à qui nous allons envoyer des messages. Par la suite nous introduirons d'autres agents qui seront capables d'interagir entre eux.
Pour envoyer un message à un agent il faut utiliser la commande **Send** :

```
{Send Agent message}
```

Remarque : La syntaxe en Oz (le langage que nous utilisons pour écrire nos programmes) impose d'utiliser des lettres majuscules pour commencer les noms de procédures (ici : Send ; nous aborderons le concept de procédure dans un cours ultérieur) ainsi que pour les noms de variables (ici : Agent et Message ; nous aborderons ce concept de variable un peu plus tard dans ce cours). N'oubliez donc pas de bien respecter les lettres minuscules ou majuscules lors de début de mots !

Agents prédéfinis

Il existe 2 agents prédéfinis dans le logiciel : Browser et Turtle.

L'agent Browser L'agent **Browser** est un agent qui permet d'afficher dans la fenêtre d'affichage (le Browser) ce que vous lui demandez.

```
{Send Browser coucou}   % affiche 'coucou' dans l'afficheur
```

Remarque : Le "%" indique ici qu'il s'agit d'un commentaire. Dans vos programmes vous pourrez écrire des lignes de commentaires de la même façon. Ainsi, ces lignes ne seront pas prises en compte lors de l'exécution du programme.

L'agent Turtle L'agent **Turtle** est la tortue visible au milieu de l'écran. Voici la liste des messages compris par cet agent :

{Send Turtle message}

$\{Send\ Turtle\ forward(X)\}$	demande à la tortue d'avancer de X pixel(s)
$\{Send\ Turtle\ penUp\}$	demande à la tortue de lever son crayon
$\{Send\ Turtle\ penDown\}$	demande à la tortue de baisser son crayon
$\{Send\ Turtle\ left(X)\}$	demande à la tortue de tourner vers la gauche d'un angle de X degrés ($0 \leq X \leq 360$)
$\{Send\ Turtle\ right(X)\}$	demande à la tortue de tourner vers la droite d'un angle de X degrés ($0 \leq X \leq 360$)
$\{Send\ Turtle\ color(C)\}$	demande à la tortue de changer la couleur de son crayon (C = green, red, blue, yellow, black, purple ou pink)

TAB. A.1 – Syntaxe des messages

La tortue peut, par exemple, se déplacer de la façon suivante :

```
{Send Turtle penDown}
{Send Turtle forward(100)}
```

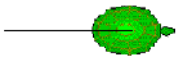

FIG. A.1 – forward(100)

La tortue s'oriente de la façon suivante :

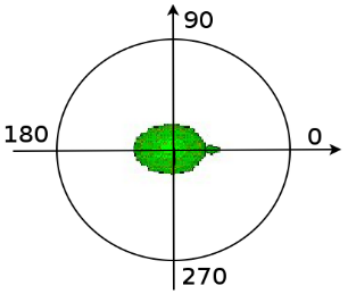

FIG. A.2 – Orientation de la tortue

A.1.3 L'interface

Voici l'interface que nous allons utiliser lors de nos séances d'exercices :
Il y a donc 3 grandes parties dans cette interface :

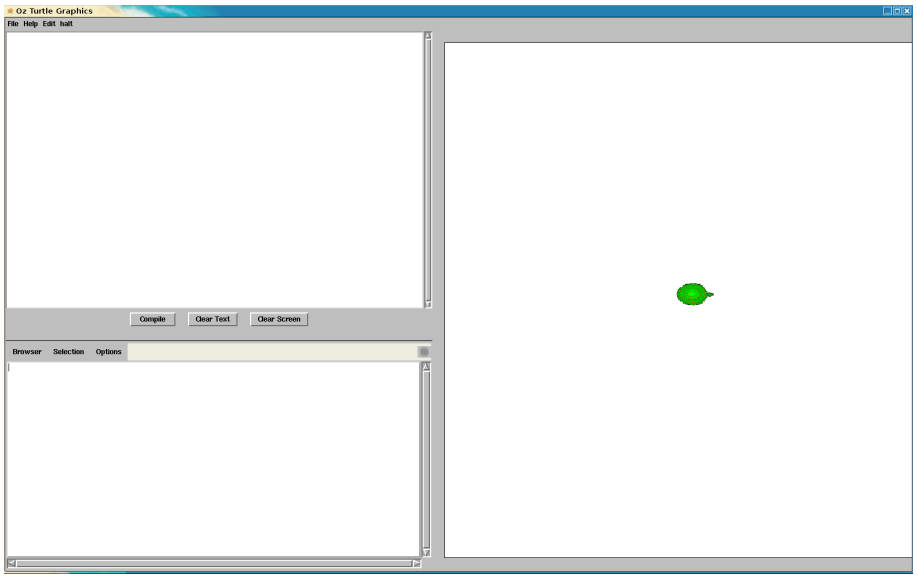

FIG. A.3 – L'interface

1. le notepad
2. le Browser
3. la feuille de dessin

Le notepad

FIG. A.4 – Le notepad

Le notepad est l'endroit où nous allons écrire les messages qui seront envoyés à l'agent. En dessous du notepad se trouvent un certain nombre d'informations et de boutons

1. Line et Char : ces deux informations indiquent la ligne à laquelle se trouve le curseur ainsi que le caractère dans la ligne ;
2. Compile : en appuyant sur ce bouton la ligne de commande écrite dans le notepad va être vérifiée. Si cette ligne est correcte et si le message à destination de l'agent est correct, la ligne sera acceptée et le message sera envoyé à l'agent. Par contre, si la ligne de commande ou le message ne sont pas valides, l'instruction sera refusée et il y aura une erreur.
3. Clear Screen : remet le monde de l'agent dans son état initial.

Le Browser

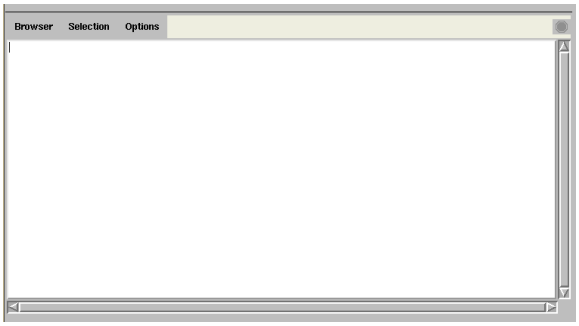

FIG. A.5 – La fenêtre d'affichage

L'agent Browser permet l'affichage de différents éléments dans la fenêtre d'affichage (le Browser) en lui envoyant un message {*Send Browser* ...}.

```
{Send Browser coucou}  % coucou

X = 2                  % on assigne la valeur 2 à la variable X
{Send Browser X}       % 2
```

Définition : Les variables sont des raccourcis vers des valeurs stockées en mémoire. L'identificateur X est le nom donné à la nouvelle variable. Un identificateur commence toujours par une lettre majuscule, suivie de lettres majuscules, minuscules, ou de chiffres. La valeur 2 est stockée en mémoire et peut être récupérée grâce à l'identificateur X.
Une variable peut être *locale* ou *globale*. Si une variable est locale, elle ne sera définie que dans une portion de code, et ne sera donc pas accessible dans le reste du code. Par contre si la variable est globale, elle pourra être accédée à partir de n'importe quel endroit dans le code. Ces deux concepts de *variablelocale* et *variableglobale* seront illustrés ultérieurement.
Une variable peut également être liée ou non. Lorsque l'on déclare une variable

```
X
```

cette variable n'est pas encore liée. Pour qu'elle soit liée, il faut lui assigner une valeur

```
X = 3
```

Une fois qu'une variable est liée, elle l'est jusqu'à la fin du programme (ou, s'il s'agit d'une variable locale, jusqu'à la fin de la portion de code dans laquelle elle est définie). On ne peut donc plus lui assigner une autre valeur.
Il est possible de déclarer une variable, et de la lier par la suite seulement

```
X               % déclaration de la variable X
{Send Browser X}     % le browser n'affiche aucune valeur car X n'est pas
                % encore lié. Il affiche seulement "_" (ou "X")
{Delay 2000}    % introduit un délai de 2000 micros seconde soit 2 secondes
X = 2           % la valeur 2 est assignée à X. Le "_" (ou "X")affiché par
                % le browser est remplacé par 2
```

Remarques :
- la procédure {*Delay* 1000} permet de faire une pause de 1000 millisecondes (1 seconde) dans le programme. Le programme s'interrompt donc pendant 1 seconde et reprend ensuite son exécution là ou il en était ;
- une suite de caractères qui ne commence PAS par une majuscule ne sera pas une variable. Il s'agira alors d'un **atome**. Un atome peut également être une suite de caractères entourée de simple guillemets (ex : '### coucou ###'), elle peut dans ce cas commencer par une Majuscule ;
- La fenêtre d'affichage permet également la gestion des erreurs. S'il y a, par exemple, une faute de frappe dans une instruction, un message concernant cette erreur apparaîtra lors de la vérification de l'instruction ;

La feuille de dessin

La feuille de dessin est le monde dans lequel la tortue évolue. Ce monde permettra de visualiser l'effet des messages envoyés à l'agent.

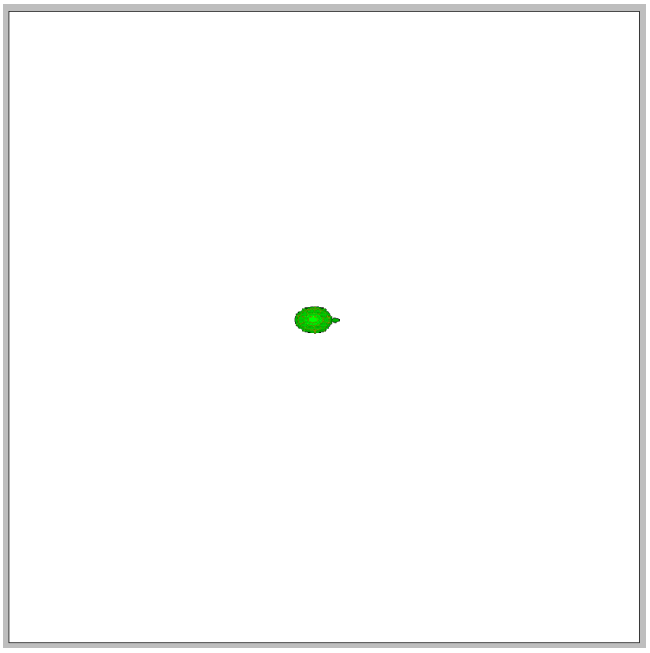

FIG. A.6 – La feuille de dessin

La feuille de dessin est un tore, c'est-à-dire que lorsque l'agent dépasse une des limites de l'espace de dessin, il réapparaît de l'autre côté. Ce système permet de ne pas couper les dessins en plein milieu.

Il nous reste alors la barre de menu qui est assez explicite. Nous y retrouvons les fonctionnalités "New", "Save", "Open", "Help" comme dans beaucoup d'autres programmes.

Il y a néanmoins certaines choses qui valent la peine d'être observées.
Tout d'abord, le menu "Edit" contient l'option "Delay", qui permettra de déterminer le temps que va prendre l'agent entre chacune des instructions qui lui seront envoyées. Un plus grand *delay* ralentira donc l'agent, et permettra de mieux voir ce qu'il fait.
Le menu "Kill" lui permet de tuer les agents. Ce menu s'avérera utile lorsque par mégarde vous aurez envoyé à un agent toute une série d'instructions qui risquent de prendre beaucoup de temps, ou qui risquent même de ne jamais s'arrêter !
Kill All Turtles servira, comme son nom l'indique, à arrêter les agents turtle.
Kill All Metronomes servira dans un cours ultérieur.

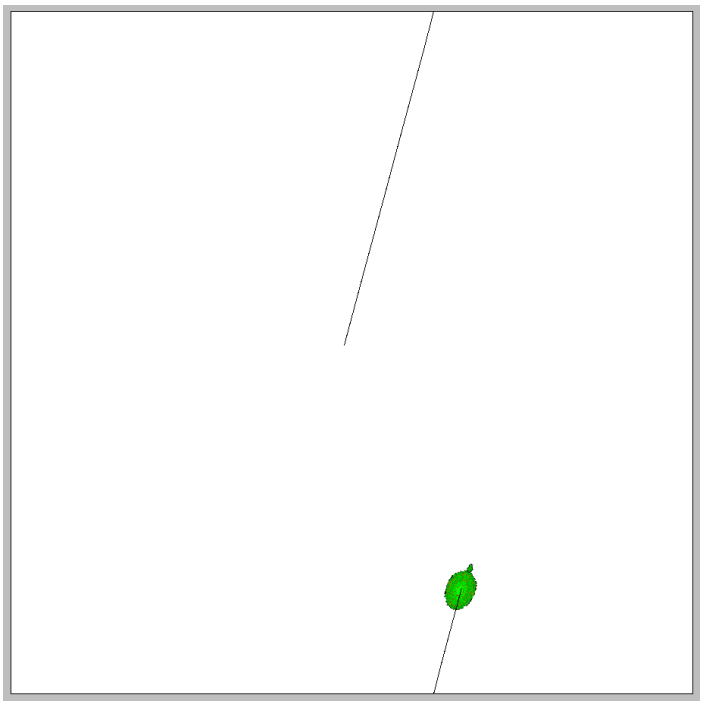

FIG. A.7 – Tore

FIG. A.8 – Menu

Instructions

Définition : Une instruction est une ligne de code qui permet de réaliser une certaine tâche.

Pour communiquer avec un ordinateur par exemple, tout se fait grâce à un certain nombre d'instructions.
Lorsque nous allons communiquer avec l'agent Turtle, nous allons également lui envoyer des instructions. Par exemple

```
{Send Turtle forward(100)}
```

est une instruction qui envoie à l'agent Turtle le message $forward(100)$.
Ce n'est évidemment pas pratique d'écrire pour chaque tâche qui doit être effectuée une instruction séparément.

Pour cela, nous allons par la suite voir des optimisations que sont les "fonctions" et les "procédures", qui permettront d'envoyer en seulement quelques instructions un grand

nombre de tâches à l'agent.

Exemples :

```
{Send Turtle penDown}
{Send Turtle left(90)}
{Send Turtle forward(100)}
{Send Turtle right(30)}
{Send Turtle forward(100)}
{Send Turtle right(120)}
{Send Turtle forward(100)}
{Send Turtle right(120)}
{Send Turtle forward(100)}
{Send Turtle left(135)}
{Send Turtle forward(141)}
{Send Turtle left(135)}
{Send Turtle forward(100)}
{Send Turtle left(135)}
{Send Turtle forward(141)}
{Send Turtle left(135)}
{Send Turtle forward(100)}
```

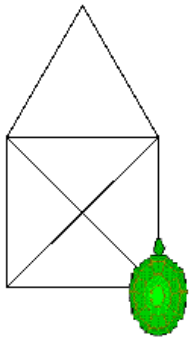

FIG. A.9 – Maison

```
{Send Turtle penDown}
{Send Turtle forward(100)}
{Send Turtle left(90)}
{Send Turtle forward(100)}
{Send Turtle left(90)}
{Send Turtle forward(100)}
{Send Turtle left(90)}
{Send Turtle forward(100)}
{Send Turtle left(90)}
```

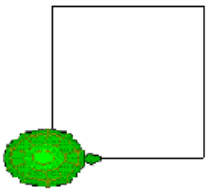

FIG. A.10 – Carré

Exercices

Pour lancer le programme il suffit d'ouvrir un terminal, de vous placer dans le répertoire contenant le programme et de taper la ligne de code suivante :

```
ozengine TurtleGraphics.oza
```

Passons aux exercices :

1. Dessinez un triangle de côté 80.
2. Dessinez le même triangle de côté 100 mais d'une autre couleur.
3. Essayez de trouver d'autres formes simples à dessiner.

A.2 Cours 2 : Procédures - Micromonde 2

A.2.1 Les procédures

Définition : On appelle procédure un sous-programme qui permet d'effectuer un ensemble d'instructions par un simple appel de la procédure. Une procédure peut être vue comme une suite d'instructions à laquelle on a donné un nom, un peu comme une recette de cuisine. Pour définir une procédure, il faut :

1. indiquer par le mot clé *proc* que l'on veut définir une nouvelle procédure ;
2. donner un nom à la procédure ;
3. marquer la fin de la procédure par le mot clé *end*.

Exemple :

```
proc{Carre}
   {Send Turtle penDown}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
   {Send Turtle forward(100)}
   {Send Turtle left(90)}
end

{Carre}
```

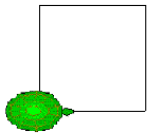

FIG. A.11 – Carré de côté 100

Une fois la procédure définie, on l'appelle par son nom. L'appel de la procédure va exécuter toutes les instructions à l'intérieur de celle-ci, ce qui donnera le dessin de la figure 1.

A.2.2 Les arguments

Il est possible de passer des arguments à une procédure, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la procédure puisse effectuer des opérations sur ces arguments, ou grâce à ces arguments. Le passage d'arguments à une procédure se fait en écrivant les arguments séparés par un espace suivant immédiatement le nom de la procédure, toujours entre les accolades.
Lors de l'appel de la procédure, le nombre et le type d'arguments doit correspondre au nombre et au type d'arguments dans la déclaration de la procédure. Sinon il y aura une erreur lors de la compilation...
Les arguments sont uniquement définis dans la procédure, ce sont donc des variables locales qui n'existent pas en dehors de la procédure. On dit que leur **portée** est limitée à la

procédure.

A.2.3 La procédure *Carré* avec un argument

Nous allons maintenant modifier la procédure *Carre* définie précédemment en lui ajoutant un argument X qui sera la taille d'un côté du carré. X remplacera le 100 dans le message $forward(X)$. Ainsi il sera possible de créer des carrés de taille différente.

```
proc{Carre X}
   {Send Turtle penDown}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
   {Send Turtle forward(X)}
   {Send Turtle left(90)}
end

{Carre 200}
{Carre 100}
{Carre 50}
```

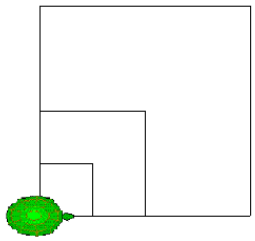

FIG. A.12 – Carré avec argument

La procédure *Repeat*

Une procédure particulière mais néanmoins fort importante est la procédure Repeat.

$$\{Repeat\ N\ P\}$$

Cette procédure permet de répéter N fois la procédure P. Souvent il est plus efficace de répéter certains blocs d'instructions plutôt que de les écrire plusieurs fois. Grâce à cette procédure nous pourrons réaliser par la suite de surprenants dessins.

La procédure *Cercle*

Un second exemple est la procédure *dessiner un cercle* :
Pour dessiner un cercle, il suffit de demander à la tortue d'avancer d'un petit pas, de tourner d'un petit angle, d'avancer d'un petit pas, de tourner d'un petit angle, et ainsi de suite jusqu'à avoir tourné de 360° au total.
Pour ce faire, nous allons bien entendu utiliser la procédure $\{Repeat\ N\ P\}$, où N est le nombre de fois que l'on veut répéter la procédure P. Il faudra répéter ces 2 actions de manière à ce que la tortue fasse au total un angle de 360°.
Nous définissons tout d'abord une procédure nommée *Cercle* qui fera avancer et tourner

la tortue. Ensuite cette procédure est répétée 360 fois de manière à dessiner un cercle.
La procédure {*Repeat N P*} prend donc 2 arguments : N et P. Il faut tout d'abord lui passer pour N le nombre 360 (degrés) et ensuite pour P le nom donné à la procédure : *Cercle*.
Attention ! il ne faut pas oublier de demander à la tortue de baisser le crayon pour voir le cercle dessiné !

```
proc{Cercle}
   {Send Turtle forward(1)}
   {Send Turtle left(1)}
end

{Send Turtle penDown}
{Repeat 360 Cercle}
```

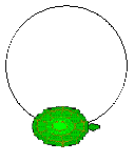

FIG. A.13 – Cercle

Une autre façon de dessiner un cercle est en utilisant la procédure *Repeat* :
Rappelons que {*Repeat N P*} répète N fois la procédure P. Il est possible de dessiner le même cercle en créant la procédure à l'intérieur du "Repeat". P n'est donc plus le nom de la procédure qui dessine le cercle (un lien vers cette procédure) mais c'est la procédure qui définit le cercle. Le mot clé "proc" est toujours utilisé, mais il n'est plus nécessaire de lui donner un nom puisque la procédure ne sera pas appelée explicitement, d'où la notation *proc*{$}. Le $ défini un nom sans importance pour la procédure. La notation suivante est alors utilisée : *proc*{$} ... *end*. On peut également voir ces 2 mots clés comme de simples délimiteurs des instructions à répéter.

```
{Send Turtle penDown}
{Repeat 360 proc{$} {Send Turtle forward(1)} {Send Turtle left(1)} end}
```

L'exécution de ce code donne le même dessin que la figure 3.

A.2.4 Dessin procédural

Nous allons présenter ici quelques exemples de dessins que vous pouvez réaliser grâce aux connaissances accumulées.

Dessiner une fleur

Voici pour commencer le dessin d'une fleur. Ce dessin est un exemple assez complet utilisant les concepts simples expliqués jusqu'à présent. Pour dessiner la fleur nous aurons besoin de différentes choses :

1. une procédure {*QCercle*} qui va dessiner un quart de cercle (pour former les pétales) ;

   ```
   proc{QCercle}
      {Repeat 45 proc{$} {Send Turtle forward(2)} {Send Turtle left(2)} end}
   end
   ```

remarquez que l'on répète 45 fois un $left(2)$ ce qui fait bien 90° ;

2. une procédure $\{Petale\}$ qui va dessiner un pétale de la fleur en utilisant la procédure $\{QCercle\}$. Un pétale n'est rien d'autre que 2 quarts de cercle ;

```
proc{Petale}
   {Repeat 2 proc{$} {QCercle} {Send Turtle left(90)} end}
end
```

3. une procédure $\{Fleur\}$ qui va dessiner la fleur. La fleur est dessinée ici comme un ensemble de 10 pétales (d'où le $right(36)$ car $10 * 36 = 360$) ;

```
proc{Fleur}
{Repeat 10 proc{$} {Petale} {Send Turtle right(36)} end}
end
```

4. une procédure $\{Plante\}$ qui dessinera la fleur accompagnée d'une tige ;

Comprenez bien les 3 procédures données car elles vont vous servir à dessiner la plante. N'oubliez pas qu'une plante n'est pas noire donc il faut également ajouter des couleurs ! Voici la ligne de code qui permet de demander à la tortue de changer la couleur de son trait : $\{Send\ Turtle\ color(green)\}$.
Vous devriez obtenir un dessin semblable à celui-ci :

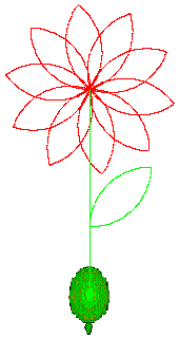

FIG. A.14 – Plante

Dessiner un oeil

Dessiner un oeil à la main peut sembler fort compliqué, mais nous allons ici vous montrer comment réaliser ce dessin de manière fort simple. Il suffit de dessiner une multitude de carrés en tournant chaque carré d'un petit angle de manière à faire un tour complet. Il suffit de deux procédures :

1. une première procédure pour dessiner un carré (nous choisirons un côté de longueur 80) ;
2. une seconde procédure qui va répéter le dessin d'un carré et modifier l'angle de dessin afin de faire un tour complet.

```
proc{Carre}
   {Repeat 4 proc{$} {Send Turtle forward(80)} {Send Turtle right(90)} end}
end

proc{Oeil}
   {Send Turtle penDown}
   {Repeat 180 proc{$} {Carre} {Send Turtle right(2)} end}
end

{Oeil}
{Send Turtle penUp}        % on déplace la tortue une fois que le dessin est fini
{Send Turtle forward(200)} % pour bien visualiser le dessin
```

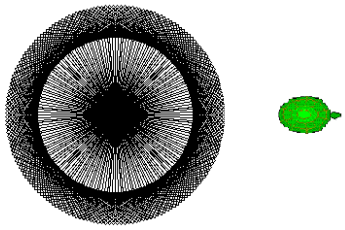

FIG. A.15 – Oeil

Dessiner un donut

Un autre exemple de dessin est celui du donut. La façon de procéder ressemble fort à celle de l'oeil. Il suffit à nouveau de deux procédures :

1. une première procédure pour dessiner un cercle ;
2. une seconde procédure pour répéter le dessin du cercle, un léger déplacement et le changement de l'angle pour chaque cercle. Un conseil, ce dessin prend beaucoup de temps pour la tortue, veillez donc à réduire le délai dans le programme.

```
proc{Cercle}
   {Repeat 90 proc{$} {Send Turtle forward(2)} {Send Turtle right(4)} end}
end

proc{Donut}
   {Repeat 180 proc{$} {Cercle} {Send Turtle forward(4)} {Send Turtle right(2)} end}
end

{Send Turtle penDown}
{Donut}
```

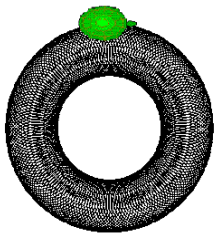

FIG. A.16 – Donut

A.2.5 Exercices

1. Ecrivez une procédure pour dessiner une maison.
   Indice : vous pouvez utiliser le code de la maison fourni dans le cours 1 pour réaliser la procédure.
2. Dessinez un carré de taille fixe en utilisant la procédure $\{Repeat\ N\ P\}$;
3. Modifiez votre programme de manière à pouvoir choisir la taille du carré.
4. Modifiez le programme de l'oeil (3.2) de manière à ce que l'iris soit bleu.
5. Réalisez la plante à l'aide des procédures fournies dans ce cours.

A.3 Cours 3 : Opérateurs et récursion - Micromonde 2

A.3.1 Les opérateurs

Avant de passer à des dessins et des concepts plus compliqués, il est urgent d'aborder certaines spécificités du langage Oz.

Commençons par parcourir la liste des opérateurs arithmétiques :

$+$	addition
$-$	soustraction
$*$	multiplication
div	division entière
/	division rationnelle
mod	modulo (reste d'une division entière)

TAB. A.2 – Opérateurs arithmétiques

Voici quelques exemples simples pour illustrer ces concepts :

```
{Send Browser 1+1}          % 2
{Send Browser 1-1}          % 0
{Send Browser ~2*2}         % -4
{Send Browser 3 div 2}      % 1
{Send Browser 3.0/2.0}      % 1.5
{Send Browser 4 mod 3}      % 1
```

Remarques importantes :

1. Dans le langage Oz il existe deux types de nombres : les nombres entiers appelés *Integer* et les nombres à virgule appelés *Float*. La division rationnelle ne peut s'effectuer que sur des *Float* et pas sur des *Integer*. Par exemple {*Send Browser* 2/1} générera une erreur car ni 2 ni 1 ne sont de type *Float*. {*Send Browser* 2/1.0} ne fonctionnera pas non plus, car dans une division rationnelle les deux membres concernés par la division doivent posséder une virgule. Il faudra donc écrire {*Send Browser* 2.0/1.0}.
2. Pour écrire -1 en langage Oz, on écrit ~1.

Continuons avec la liste des opérateurs binaires les plus courants dont vous allez sans doute avoir besoin lors des TP :

==	égalité
\ =	différence
<	strictement plus petit
=<	plus petit ou égal
>	strictement plus grand
>=	plus grand ou égal

TAB. A.3 – Opérateurs binaires

Voici quelques exemples simples pour illustrer ces concepts :

```
{Send Browser 1==1}          % true
{Send Browser 1\=1}          % false
{Send Browser 3 > 2}         % true
{Send Browser 3 =< 2}        % false
```

A.3.2 Conditionnel : la commande *if*

La commande if est une commande conditionnelle, c'est-à-dire que cette commande va permettre de poser une condition dans le programme. Si cette condition est remplie le programme exécutera une série d'instructions, sinon il en exécutera d'autres.

```
if <x1> then
   <s1>
elseif <x2> then
   <s2>
else
   <s3>
end
```

Si la condition $x1$ est remplie, $s1$ est exécuté. Sinon, si la condition $s2$ est remplie, $s2$ est exécuté. Sinon, $s3$ sera exécuté.

Exemples :

1. ```
   Y = 3
   if Y \= 3 then
      {Send Browser erreur}
   else
      {Send Browser correct}
   end
   ```
   ```

2.
```
X = 3
if X > 3 then
   {Send Browser plusGrand}
elseif X < 3 then
   {Send Browser plusPetit}
else
   {Send Browser egal}
end
```

Définition d'un variable à l'intérieur d'une fonction/procédure

Il est possible de créer une variable à l'intérieur d'une fonction/procédure. Pour ce faire, on utilise le mot clé *in*.

```
%exemple
proc{Test X}
   if(X < 3) then Y in
      Y = X
      {Send Browser Y}
   end
   %ici Y n'existe plus !!!
   %{Send Browser Y} générera une erreur !
end

{Test 2}
```

La variable Y existe à partir de sa définition jusqu'au prochain mot clé *end*. Sa portée est donc limitée entre le *in* et le *end*. Cette variable n'existe bien entendu pas non plus à l'extérieur de la procédure.

A.3.3 Pattern matching : la commande *case of*

La commande *case of* permet de comparer deux éléments entre eux, et d'exécuter la suite du programme en fonction du résultat de cette comparaison.

```
case <x>
   of <l1> then <s1>
   [] <l2> then <s2>
   else <s3>
end
```

On compare x d'abord à $l1$. Si les deux éléments correspondent, $s1$ sera exécuté. Sinon, on passe à la prochaine ligne. x sera alors comparé à $l2$ et ainsi de suite.
Dans tous les autres cas, c'est-à-dire si aucune correspondance n'est trouvée, $s3$ sera exécuté.

Exemple :

```
case Msg
   of coucou then
      {Send Browser coucou}
   [] salut then
      {Send Browser salut}
   else
      {Send Browser auRevoir}
end
```

A.3.4 La Récursivité

On dit qu'une procédure est récursive si elle s'appelle elle-même. La récursivité permet, tout comme le *for*, de répéter plusieurs fois une action.
Dans la récursivité on distingue souvent 2 cas :

1. le cas de base, dans lequel la procédure se termine ;
2. le cas récursif, dans lequel se trouve l'appel récursif afin de boucler.
   Attention ! toutes les instructions écrites après l'appel récursif seront sauvegardées sur la pile (stack) pour pouvoir être exécutées par la suite !

Un exemple simple

Voici un exemple qui consiste à faire plusieurs *forward* les uns après les autres en utilisant la récursivité :

```
proc{MoveForward X}
   if(X == 0) then skip            % fin de la procédure
   else
      {Send Turtle forward(25)}    % avance de 25
      {Delay 1000}                 % attend pendant 1 sec
      {MoveForward X-1}            % appel récursif de la procédure
   end
end

{Send Turtle penDown}
{MoveForward 3}                    % appel de la procédure
```

Remarques :

1. X représente le nombre de fois que l'on souhaite exécuter un *forward*, c'est-à-dire le nombre d'appels récursifs qui seront faits ;
2. Le mot clé *skip* est utilisé pour quitter une procédure. Cette instruction permet donc de sortir de la procédure et le programme se termine ;

Comment fonctionne ce programme ?
– on appelle le programme avec '3' comme argument.

– le programme teste si X est différent de 0. Ce n'est pas le cas car X vaut 3. Le programme entre donc dans le *else* et demande à la tortue d'avancer de 25. Le {*Delay* 1000} est là uniquement pour permettre de visualiser la récursivité. Une fois la seconde écoulée, le programme rappelle la procédure *Forward* en décrémentant l'argument (c'est-à-dire X) de 1 ;
– X vaut maintenant 2. 2 étant différent de 0, le programme entre à nouveau dans le *else* et la tortue avance de 25. Un nouvel appel récursif est ensuite effectué avec $X - 1$;
– X vaut maintenant 1. 1 étant toujours différent de 0, la tortue avance à nouveau de 25 et la procédure est rappelée avec $X - 1$;
– X vaut maintenant 0. Le programme entre donc dans la condition $if\ (X == 0)$ et tombe sur l'instruction skip. Le programme quitte donc la procédure et se termine.

Dessiner une spirale

Voyons maintenant comment faire des dessins en utilisant la récursivité.

Commençons par un dessin de spirale :

```
proc{Spirale X}
   if(X < 100) then
      {Send Turtle forward(X)}
      {Send Turtle right(45)}
      {Spirale X+1}
   else skip
   end
end

{Send Turtle penDown}
{Spirale 1}
```

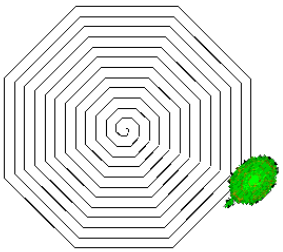

FIG. A.17 – Spirale 45°

Observez ce qui se passe lorsque l'on change l'angle *right*(45) en *right*(70). Vous devriez obtenir le dessin suivant :

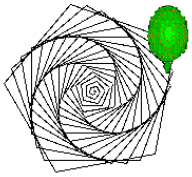

FIG. A.18 – Spirale 70°

Dessiner un arbre

Le dessin d'un arbre est un exemple typique de dessin qui utilise la récursion. L'objectif est de dessiner un arbre tel que celui de la figure 5.

FIG. A.19 – Arbre

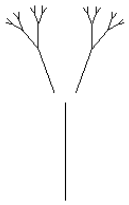

FIG. A.20 – Arbre décomposé

La difficulté consiste à identifier l'arbre comme un problème récursif. Voyez-vous les différents sous-problèmes ? L'arbre consiste en un tronc avec deux petites branches attachées, auxquelles on attache encore 2 petites branches, et ainsi de suite.
Il existe deux approches pour limiter le nombre de branches de l'arbre.

Première approche La première approche va nous permettre d'exprimer la profondeur, c'est-à-dire le nombre d'étages de l'arbre, lors de l'appel de la procédure récursive. La procédure *Arbre* aura alors 2 arguments :

- *Depth*, la profondeur de l'arbre ;
- *Size*, la longueur des branches.

```
proc{Tree Size Depth}
   if(Depth == 0) then skip          % fin de la procédure
   else
      {Send Turtle forward(Size)}
      {Send Turtle left(20)}
      {Tree (Size div 2) Depth-1}    % appel récursif pour la partie gauche
      {Send Turtle right(40)}
      {Tree (Size div 2) Depth-1}    % appel récursif pour la partie droite
      {Send Turtle left(20)}
      {Send Turtle left(180)}
      {Send Turtle forward(Size)}
      {Send Turtle left(180)}
   end
end

{Send Turtle left(90)}
{Send Turtle penDown}
{Tree 200 5}
```

La tortue commence donc par dessiner le tronc. Ensuite elle pivote à gauche et l'appel récursif est fait avec $Size\ div\ 2$ pour que les branches soient plus petites et $Depth - 1$ pour indiquer que le premier étage est terminé.
Afin de comprendre comment ce programme fonctionne, examinons l'ordre dans lequel les instructions sont exécutées.
Supposons ici que l'on exécute le programme $\{Tree\ 100\ 2\}$ afin de simplifier l'explication. La première colonne représente les instruction exécutées par la tortue et la deuxième représente la pile d'instructions :

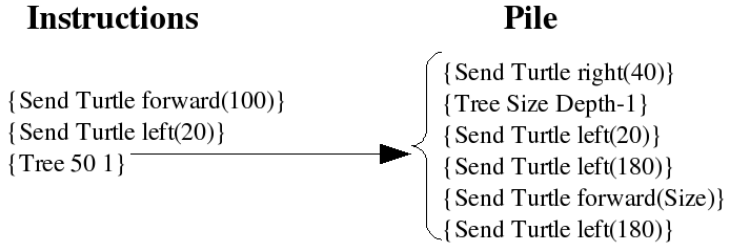

FIG. A.21 – Pile

La tortue exécute donc un $forward$ suivi d'un $left$. Ensuite il y a un appel récursif, les instructions qui n'ont pas pu être exécutées (c'est-à-dire celles qui suivent l'appel récursif) sont sauvées sur la pile et l'appel récursif est fait. Le programme revient donc au début de la procédure.
Le scénario est le même : les instructions qui suivent l'appel récursif sont sauvées sur la pile. La pile est une pile LIFO (Last in, First out, un peu comme une pile d'assiettes). Les instructions mises en dernier sur la pile sont au sommet de cette pile. L'appel récursif est fait :

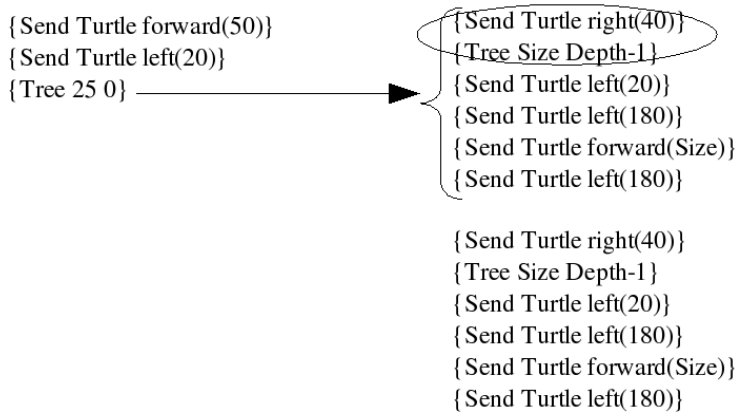

FIG. A.22 – Pile

Le programme se retrouve à nouveau en début de procédure, et cette fois-ci $Depth\ ==$

0 est vrai ! Le programme tombe sur l'instruction *skip* et la procédure se termine. A ce moment là les instructions restantes sur la pile vont être exécutées. La tortue va donc exécuter les 2 instructions entourées : *right* et l'appel récursif.
Après avoir exécuté le *right* et l'appel récursif qui étaient sur la pile, le programme revient en début de procédure et cette fois encore $Depth == 0$ est vrai ! On tombe sur l'instruction *skip* et la procédure se termine. Les instructions qui restent sur la pile sont exécutées. La tortue va donc exécuter les 6 instructions entourées :

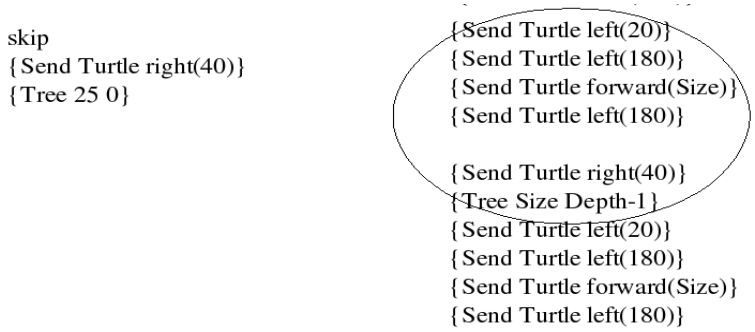

FIG. A.23 – Pile

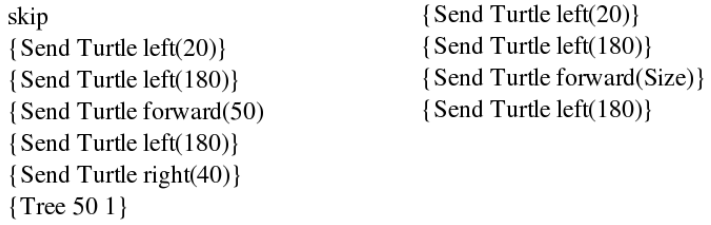

FIG. A.24 – Pile

Après avoir exécuté les instructions et l'appel récursif, le programme se retrouve au début de la procédure. Cette fois-ci $Depth == 0$ est faux. La tortue exécute alors à nouveau un appel récursif :

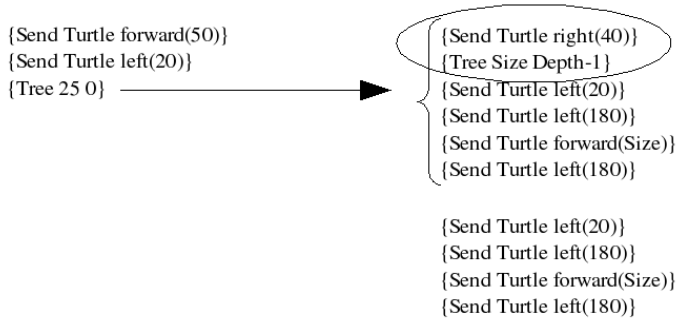

FIG. A.25 – Pile

Une fois que le programme arrive à l'appel récursif il faut sauver sur la pile les instructions qui suivent cet appel. Après l'appel récursif le programme arrive au début de la procédure et $Depth == 0$ est vrai ! On tombe sur l'instruction *skip* et la procédure se termine. Les instructions qui sont sur la pile sont alors exécutées :

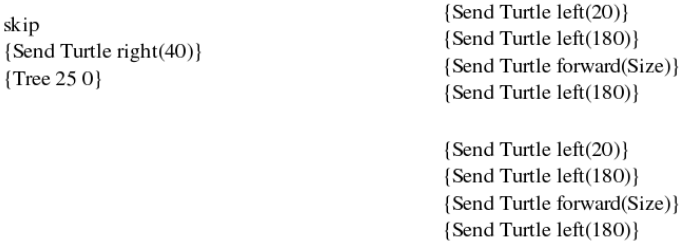

FIG. A.26 – Pile

Après avoir exécuté *right* et l'appel récursif, $Depth == 0$ est vrai. La procédure se termine donc et les instructions restantes sur la pile sont exécutées :

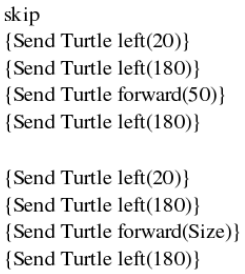

FIG. A.27 – Pile

OUF ! C'est fini ! En reprenant la liste des instructions exécutées par la tortue on

peut constater qu'elle dessine bien un arbre avec 2 branches. Si maintenant la procédure {*Tree* 200 5} est exécutée, on obtiendra le même arbre que la Fig. 5

Deuxième approche La deuxième approche pour dessiner le même arbre consiste à garder le nombre de branches de l'arbre supérieur à un minimum raisonnable, par exemple 4. Voici comment procéder :

```
proc{Tree Size}
   if(Size < 4) then skip
   else
      {Send Turtle forward(Size)}
      {Send Turtle left(20)}
      {Tree (Size div 2)}
      {Send Turtle right(40)}
      {Tree (Size div 2)}
      {Send Turtle left(20)}
      {Send Turtle left(180)}
      {Send Turtle forward(Size)}
      {Send Turtle left(180)}
   end
end

{Send Turtle left(90)}
{Send Turtle penDown}
{Tree 200}
```

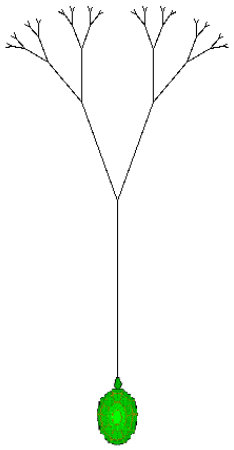

FIG. A.28 – Arbre

A.3.5 Exercice

En vous basant sur le principe utilisé au point 5.2, dessinez un soleil. Voici quelques indices pour vous guider :

1. Créer une procédure *Soleil* à 2 arguments, *Longueur* et *Angle* : {*Soleil Longueur Angle*} ;
2. Tester si la longueur est plus petite que 400, par exemple (400 sera alors la longueur maximale du trait) ;
3. Cas récursif : avancer de "Longueur", tourner de "Angle" degrés (l'angle doit être légèrement supérieur à 180° de manière à dessiner les rayons du soleil, 185° par exemple), rappeler la procédure en incrémentant la longueur (de 2 par exemple) ;
4. Cas de base : sortir de la procédure.

Vous devriez obtenir un dessin semblable à celui-ci :

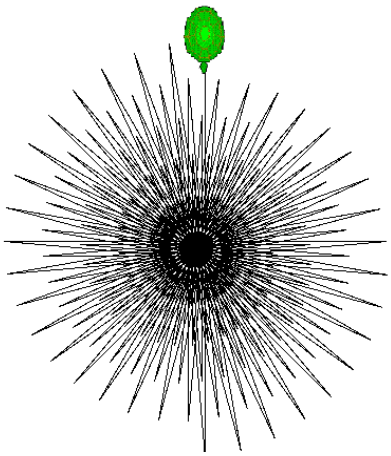

FIG. A.29 – Soleil

A.4 Cours 4 : Multi-Turtles - Micromonde 3

A.4.1 Agent : Rappel de la définition

Comme vu précédemment lors du premier cours, un agent est une entité avec laquelle il est possible de communiquer grâce à un certain nombre de messages bien définis qui peuvent être compris par cet agent.

Un agent peut donc recevoir des messages et agir en conséquent. Il peut également envoyer lui-même un message à un autre agent qui réagira. Il peut donc y avoir plusieurs agents qui communiquent entre eux et qui interagissent.
Jusqu'à présent nous nous sommes contentés d'un seul agent, la tortue, à qui nous avons envoyé des messages déjà prédéfinis.
Nous allons maintenant examiner d'autres aspects concernant les agents : leur état, leur cerveau, la communication entre différents agents, etc.

A.4.2 Etat

Chaque agent possède son propre état qui peut être composé d'éléments variés. L'état d'un agent correspond à l'état dans lequel il se trouve à un moment donné.

Etat de la tortue

Examinons l'état de notre tortue que nous connaissons maintenant si bien :

```
state(Name pos(X Y) Direction PenDown Color)
```

L'état de la tortue est composé de 5 éléments distincts :

1. son nom : la tortue présente dans le monde s'appelle 'Turtle'. Lorsque l'on crée une nouvelle tortue il est préférable lui donner un nom. Ce nom est un atome (cf cours 1) ;
2. sa position : la position est composée de deux coordonnées, X et Y. Ces coordonnées permettent de situer la tortue dans l'espace à deux dimensions qu'est le monde dans lequel elle évolue. Les coordonnées X et Y sont mises à jour lors de chaque déplacement de la tortue, c'est-à-dire lors de chaque réception d'un message de type $forward(N)$;
3. sa direction : la direction permet de mémoriser vers quelle direction la tortue est actuellement tournée. A chaque envoi de message $left(N)$ ou $right(N)$ cette direction est mise à jour ;
4. PenDown : il s'agit d'un booléen (vrai ou faux) qui indique si le crayon de la tortue est actuellement posé ou non. Le booléen passera à *vrai* lors de la réception d'un message du type *penDown*, et repassera à faux lors de la réception d'un message $penUp$;

5. et sa couleur : il s'agit tout simplement de la couleur dans laquelle les traits du crayon apparaissent à l'écran. Un choix de 7 couleurs est possible : rouge (red), bleu (blue), vert (green), jaune (yellow), noir (black), mauve (purple) et rose (pink).

L'état de la tortue peut être vu en cliquant une fois dessus à l'aide du bouton gauche de la souris.
L'état de la tortue est donc composé de plusieurs éléments. Un structure de données un peu particulière est utilisée pour le représenter : le *tuple*.

Les Tuples

Un tuple est composé d'un label suivi d'un ensemble d'éléments.
- le label est le nom donné au tuple. Dans notre cas, le tuple se nomme "*state*" ;
- les éléments du tuple sont l'ensemble des composants de l'état (nom, position, etc.).

Le fait que l'état soit un tuple représente un certain avantage. En effet, notre tuple "state" pourrait se récrire de la façon suivante :

```
Etat = state(1:Name 2:pos(X Y) 3:Direction 4:PenDown 5:Color)
```

Chaque élément est donc numéroté, en commençant à 1. Grâce à cela, les informations contenues dans le tuple peuvent être accédées très facilement.
Par exemple, pour trouver le nom de l'agent, il suffit d'écrire

```
Etat.1          % renvoie l'élément occupant la première position dans le tuple
```

Exemple :

```
Etat = state(turtle 200 300 45 true blue)

{Send Browser Etat.5}        % blue
{Send Browser Etat.3}        % 300

%Un tuple peut contenir un autre tuple
X = test(100 tuple(200 300) 400)
{Send Browser X.2}           % tuple(200 300)
{Send Browser X.2.2}         % 300
```

Obtenir l'état de la tortue dans une variable

Nous savons que nous pouvons visualiser l'état de la tortue en cliquant dessus. Mais si maintenant nous avons besoin d'avoir cet état dans une variable afin de l'utiliser dans un programme, la technique est fort différente. Le principe est le suivant :
- Définir une variable non liée (cf Cours 1) ;
- Envoyer un message à la tortue en lui demandant de mettre son état actuel dans cette variable. La tortue va lier la variable à son état ;

– On peut alors lire et utiliser cette variable.

```
X                                   % variable non liée
{Send Turtle giveYourState(X)}
{Send Browser X}                    % la variable est liée et contient
                                    % l'état de la tortue
```

A.4.3 Cerveau

Vous l'avez sans doute déjà découvert mais l'agent tortue possède une certaine intelligence, bien qu'elle soit limitée. En effet, elle ne comprend qu'un certain nombre de messages bien déterminés. Dans le cadre du cours 5 vous aurez l'occasion de créer vous mêmes vos propres agents et de définir leur propre intelligence. L'intelligence d'un agent n'est rien d'autre qu'un ensemble de messages auxquels une action est associée.

A.4.4 Création d'un nouvel agent tortue

Comment créer un nouvel agent tortue ? Voici la question à laquelle nous allons tenter de répondre dans la section qui suit.

Commençons tout d'abord par introduire un nouveau concept : les *fonctions*.

Les fonctions

Une fonction est comme une procédure si ce n'est qu'elle est introduite par le mot clé *fun* et non plus *proc*, et qu'elle renvoie un résultat. De plus, la fonction renvoie une valeur contrairement à la procédure.

Exemple :

Procédure :

```
proc {CarreP X Solution}
   Solution = X*X
end

X
{CarreP 2 X}          % la solution est conservée à l'intérieur de la
                      % procédure
{Send Browser X}      % affiche 4
```

Fonction :

```
fun {CarreF X}
   X*X
```

```
end

Y = {CarreF 2}          % la fonction CarreF renvoie une valeur qui est
                        % associée à Y
{Send Browser Y}        % affiche 4
```

La fonction NewTurtle

Pour créer une nouvelle tortue on utilise la fonction $X = \{NewTurtle\ State\}$. La fonction renvoie un nouvel agent tortue. Le nouvel agent est associé à l'identificateur X. Le cerveau de cet agent est identique à celui de la tortue d'origine. Il répond donc aux mêmes messages mais possède son propre état. Lorsque l'on crée une nouvelle tortue il faut lui passer comme argument son état initial *State*.
Attention ! il faut passer le tuple avec tous les paramètres de l'état en paramètre à la fonction. L'ordre de ces paramètres a de l'importance. Un état correct est par exemple le suivant :

$$state('maTortue'\ 200\ 200\ 0\ false\ black)$$

Cet état représentera une tortue appelée 'maTortue' positionnée en (200,200) avec une direction de 0°, un crayon levé et une couleur de crayon noire. Une fois la nouvelle tortue créée, elle est prête à recevoir des messages.

```
MaTortue = {NewTurtle state('tortue' 200 200 0 true yellow)}
proc{Squaggle}
   {Send MaTortue forward(50)}
   {Send MaTortue right(150)}
   {Send MaTortue forward(60)}
   {Send MaTortue right(100)}
   {Send MaTortue forward(30)}
   {Send MaTortue right(90)}
end
{Repeat 20 Squaggle}

{Send Turtle penDown}
{Send Turtle forward(100)}
```

FIG. A.30 – Deux tortues

A.4.5 Concurrence

La concurrence apparaît lorsque deux ou plusieurs activités indépendantes peuvent être exécutées simultanément. Il ne devrait y avoir aucune interférence entre ces activités, a moins que le programmeur ne décide qu'il est nécessaire qu'elles communiquent entre elles.

Le programme avec les deux tortues ci-dessus est déjà un programme concurrent puisque les deux tortues réalisent leur dessin simultanément et de manière indépendante.

Nous vous présentons ici un autre exemple de concurrence dans lequel quatre tortues vont apporter leur contribution à la réalisation d'un dessin commun :

```
T1 = {NewTurtle state('mat' 440 340 270 true black)}
T2 = {NewTurtle state('mat' 440 440 180 true black)}
T3 = {NewTurtle state('mat' 340 440 90 true black)}
{Send Turtle penDown}
proc{Carre T X}
   if(X =< 0) then {Send T penUp}{Send T forward(200)}skip
   else
      {Send T forward(X)}
      {Send T right(91)}
      {Carre T X-3}
   end
end

{Carre Turtle 100}
{Carre T1 100}
{Carre T2 100}
{Carre T3 100}
```

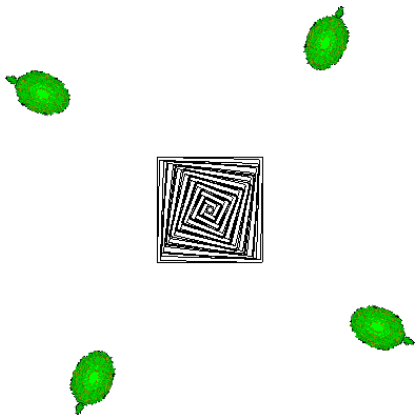

FIG. A.31 – Quatre tortues réalisant un dessin commun

A partir du moment où il y a plusieurs agents dans un programme et que ceux-ci agissent simultanément, le programme est un programme concurrent.

A.4.6 Exercice

Le but de l'exercice est de réaliser un visage très simple, comme le dessinerait un enfant, mais utilisant plusieurs tortues en concurrence. Chaque tortue réalisera une partie du visage indépendamment des autres tortues. Vous aurez besoin de sept tortues distinctes. Voici la démarche que nous vous proposons de suivre :

– Créez sept tortues, positionnez-les à des endroits stratégiques (une tortue pour la bouche, une tortue pour le nez, une tortue pour chaque oreille, une pour chaque oeil et une pour les cheveux) ;
– Créez chacune des procédures nécessaires. Vous pouvez mettre la tortue en argument de manière à rendre vos procédures générales :

```
proc{Nez T} {Send T ...} ... end
```

Pour les yeux, reprenez la procédure que vous avez réalisée au cours2 ;
– Votre dessin devrait ressembler à la figure 3.

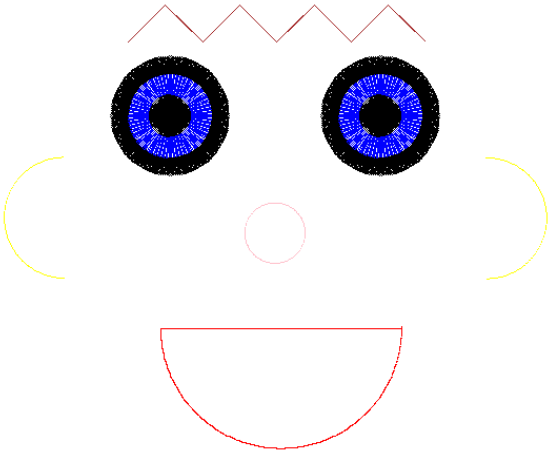

FIG. A.32 – Un visage

A.5 Cours 5 : Le temps - Micromonde 4

A.5.1 Objectif

L'objectif de cette scéance est de mettre en évidence la concurence ainsi que la communication entre différents agents. Cette scéance s'appuie sur un autre type d'agent prédéfini, le *Metronome*.

A.5.2 L'agent Métronome

Il existe un autre agent prédéfini, dans le logiciel "Oz Turtle Graphics", qui est un agent de type "Metronome". Cet agent permet d'envoyer, exactement toutes les secondes, un message à d'autres agents. Cet agent n'est rien d'autre qu'un active object tout comme la tortue. Un tel agent a donc besoin d'une fonction de transition et d'un état. Tout comme pour la tortue, la fonction de transition est déjà existante dans le logiciel. Il suffit donc pour créer un tel agent de lui fournir son état. Voici la syntaxe utilisée :

$$MyClock = \{NewMetronome\ State\}$$

La fonction "NewMetronome" prend un seul argument : State. Quel est l'état d'un tel agent ? Il envoie un message à un certain nombre d'agents. Il s'attend donc à avoir le message à envoyer et une liste d'agents auxquels il doit envoyer le message. L'état de notre nouvel agent est donc un tuple composé de deux variables :

1. la liste des agents à qui il doit envoyer le message ;
2. le message à envoyer à ces agents.

$$state([Agents]\ Msg)$$

Une nouvelle structure de données : la liste

Une liste est soit un atome *nil* représentant la liste vide, soit un tuple utilisant l'opérateur infixe | et deux arguments qui sont respectivement la **tête** et la **queue** de la liste.
Par exemple, la liste des trois premiers entiers positifs est représentée par

```
1|2|3|nil
```

Une liste qui se termine par nil peut également être représentée par tous les éléments mis entre [ ], chacun séparés par un espace et sans écrire le *nil* à la fin.

```
[1 2 3] == 1|2|3|nil
```

Nous utiliserons dans la suite du cours la notation avec les [ ]qui est bien plus simple à écrire.

Les agents dans la liste d'agents peuvent être de n'importe quel type, qu'ils soient des agents tortues ou des agents que vous avez définis vous-mêmes. Bien sûr le message qui sera envoyé doit pouvoir être compris par chacun des agents se trouvant dans la liste.

Les messages acceptés par l'agent Metronome

Voici un tableau reprenant les messages que l'on peut envoyer à un agent de type Metronome :

start	démarre le Metronome
stop	arrête le Metronome
register(*Agent*)	ajoute l'agent *Agent* à la liste des agents auxquels il faut envoyer un message
message(Msg)	définit le message Msg à envoyer aux agents
clear	réinitialise l'état du Metronome à *state*(*nil nil*) La liste d'agents est vide ainsi que le message à envoyer

TAB. A.4 – Syntaxe des messages

Prenons un exemple concret : imaginons que l'on souhaite faire avancer notre tortue de 100 pas toutes les secondes. Voici deux façons de procéder :

Première façon : on définit l'état du métronome lors de sa création.

```
{Send Turtle penDown}
MyClock = {NewMetronome state([Turtle] forward(100))}
{Send MyClock start}
```

Deuxième façon : on définit un état vide (le mot clé *nil* signifie qu'on ne met rien dans le tuple) et on définit ensuite l'agent à qui l'on souhaite envoyer le message ainsi que le message à envoyer. Ceci afin d'illustrer l'utilisation des messages *register*(*Agent*) et *message*(Msg).

```
{Send Turtle penDown}
MyClock = {NewMetronome state(nil nil)}
{Send MyClock register(Turtle)}
{Send MyClock message(forward(100))}
{Send MyClock start}
```

Pour arrêter la tortue, il faut arrêter l'agent Metronome. Pour cela, il suffit de lui envoyer un message *stop* ce qui arrête le métronome ou de cliquer sur le bouton *AllMetronome* dans le menu déroulant *Kill* du logiciel ce qui va détruire tous les métronomes créer. Si vous utiliser ce bouton, vous devrez recréer vos métronome car ils auront été détruits.

A.5.3 Exemple

```
{Send Turtle kill}  % supprime l'agent Turtle
T1 = {NewTurtle state('t1' 440 340 270 true green)}
T2 = {NewTurtle state('t2' 100 200 0 false blue)}
MyClock = {NewMetronome state(nil nil)}
```

```
{Send MyClock register(T1)}
{Send MyClock register(T2)}
{Send MyClock message(forward(100))}
{Send MyClock start}
```

A.6 Cours 6 : Les agents - Micromonde 5

A.6.1 Objectif

L'objectif de ce cours est d'apprendre à créer des agents autres que des agents tortue et des agents métronome. Ces agents auront un comportement propre, défini par le cerveau que son créateur lui attribuera. Nous nous focaliserons dans ce cours sur la création d'un agent *horloge*, qui affichera l'heure sous forme d'une montre à aiguilles.

A.6.2 Création d'un nouvel agent

Nous allons maintenant nous intéresser à la création d'un agent. Jusqu'à présent nous avons créé des agents dont la fonction de transition (c'est-à-dire son cerveau) était intégrée au logiciel. En d'autres mots, nous ne pouvions pas définir le comportement de l'agent (tortue et Metronome). Dans cette section, nous nous intéressons à la création d'agents dont le comportement sera défini à l'aide d'une fonction de transition que nous allons définir nous-mêmes. De quoi avons-nous besoin pour créer un agent ?

1. une **fonction de transition** pour définir son comportement (cerveau) ;
2. un **état initial**.

Tout comme pour la création d'une tortue, il existe une fonction permettant de créer un nouvel agent :

$$MonAgent = \{NewAgent\ FonctionTransition\ Etat\}$$

La fonction *NewAgent* renvoie le nouvel agent créé et associe ce dernier à l'identificateur *MonAgent*. Cette fonction possède deux arguments : la fonction de transition et l'état initial. La fonction de transition de l'agent (son cerveau) doit être une fonction définie par l'utilisateur. L'état initial doit être l'état dans lequel l'agent se trouvera lors de sa création. Nous allons maintenant voir comment il est possible de définir une fonction de transition.

Définir une fonction de transition pour un agent

Une fonction de transition prendra en règle générale 2 arguments :

1. **un message :** ce message correspond aux messages que reçoit l'agent. En fonction du message reçu l'agent effectuera une action. Cette technique est communément appelée "message passing" (échange de message). L'action effectuée peut être, par exemple, l'envoi d'un message à un autre agent, un déplacement ou l'affichage d'une information.

2. **un état :** il s'agit de l'état interne de l'agent. Celui-ci permet à l'agent d'avoir une mémoire. Cette mémoire peut l'aider à prendre certaines décisions ou tout simplement permet de mémoriser des informations.

Remarque : Le message peut soit être un atome, soit un tuple, ce qui permet d'envoyer beaucoup plus d'informations en une fois. L'état peut également être un tuple, ce qui permet à l'agent de mémoriser différentes informations (par exemple sa position, sa couleur, etc.)

Voici un exemple de fonction de transition qui se comporte comme un **compteur :**

```
fun{Cerveau Msg State}
   case Msg
   of tick then
      {Send Browser State+1}
      State+1
   else
      {Send Browser 'I do not understand'}
      State
   end
end
```

Comment fonctionne cette fonction ? Dès que l'agent va recevoir un message il va l'inspecter à l'aide du pattern "case of" (cf cours 3). Si le contenu correspond à un des messages connus par l'agent, alors une action prédéfinie sera exécutée. Dans notre exemple, on souhaite implémenter un petit compteur. Dès qu'un message "tick" est envoyé à notre agent compteur, celui-ci va incrémenter son état et l'afficher.

Attention ! Une fonction renvoie toujours une valeur. **L'élément que renvoie la fonction doit être le nouvel état de l'agent !**. Dans le cas d'un message tick le nouvel état est l'ancien état incrémenté. Dans le cas d'un autre message, c'est-à-dire un message que l'agent ne comprend pas, le nouvel état est le même que l'ancien (comme s'il ne s'était rien passé). En ce qui concerne la création de l'agent "Compteur", on lui passe comme fonction de transition celle que nous venons de définir et comme état initial 0. Voici l'exemple complet :

Il est bien entendu possible d'utiliser l'agent Metronome pour envoyer des messages "tick" toutes les secondes à notre agent "Compteur". Dans ce cas, il faudrait remplacer les $\{Send\ MonAgent\ tick\}$ par l'appel au Metronome que nous avons déjà vu précédemment :

```
MyClock = {NewMetronome state([Compteur] tick)}
{Send MyClock start}
```

A.6.3 Exercice : Une horloge

Pour cet exercice vous allez devoir implémenter une horloge en utilisant les concepts d'agents et d'échange de messages comme nous les avons vus plus haut. L'agent "Metronome" est évidemment indispensable pour mener à bien cet exercice. Nous mettons à votre

```
fun{Cerveau Msg State}
   case Msg
   of tick then
      {Send Browser State+1}
      State+1
   else
      {Send Browser 'I do not
       understand the message'#Msg}
      State
   end
end

Compteur = {NewAgent Cerveau 0}

{Send Compteur tick}
{Send Compteur tick}
{Send Compteur tack}
```

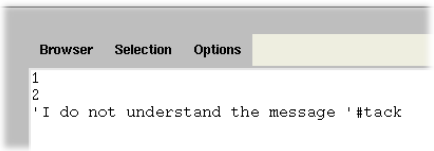

FIG. A.33 – un compteur

disposition deux fonctions qui vont vous permettre de dessiner une ligne à l'écran (afin de représenter les aiguilles) et d'effacer une ligne (afin de pouvoir faire bouger les aiguilles) :

Fonction permettant de dessiner une ligne :

$$Tag = \{DrawLine\ Size\ Direction\ Color\}$$

Cette fonction possède trois arguments :

- *Size* qui permet de définir la longueur de la ligne que l'on dessine à l'écran (en pixel) ;
- *Direction* qui permet d'orienter la ligne que l'on dessine à l'écran (en degrés). L'orientation est la convention trigonométrique classique ;
- *Color* qui permet de définir la couleur de la ligne que l'on dessine à l'écran.

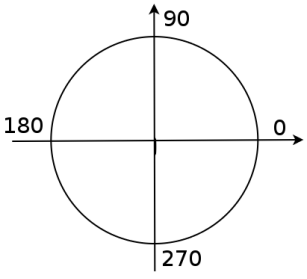

FIG. A.34 – cercle trigonométrique

Cette fonction renvoie ce que l'on appelle un "Tag" qui permet d'identifier la ligne dessinée. Ce "Tag" va permettre de retrouver une ligne qui a été dessinée et de l'effacer par

la suite.

Procédure permettant d'effacer une ligne :

$$\{DeleteTag\ Tag\}$$

Cette procédure possède comme seul argument Tag. Elle supprime l'image à l'écran identifiée par le Tag.

A.7 Projet : Programmation multi-agents - Micromonde 5

A.7.1 Objectif

L'objectif de ce projet est d'apprendre à réaliser des petits programmes multi-agents, c'est-à-dire des programmes où plusieurs agents interagissent entre eux. L'exercice de l'horloge du cours précédent était déjà un programme multi-agent. En effet, il y avait différents agents : l'agent seconde, l'agent minute et l'agent heure. Une fois arrivé à 60, l'agent seconde envoyait un message à l'agent minute et ainsi de suite.
Il existe un tas d'applications pour des programmes multi-agents, et il en existe beaucoup dans le domaine des jeux. Vous connaissez sans doute tous le jeu de ping-pong dans lequel il y a deux palettes et une balle, le but étant de renvoyer la balle à l'adversaire. D'autres jeux tels que le "Game Of Life", le "Space Invaders" sont également des applications de la programmation multi-agents. Nous vous proposons, dans ce cours, un fil conducteur pour réaliser le jeu du ping-pong.

A.7.2 Ping-Pong

La figure 1 vous donne un petit aperçu de ce à quoi vous devriez arriver.

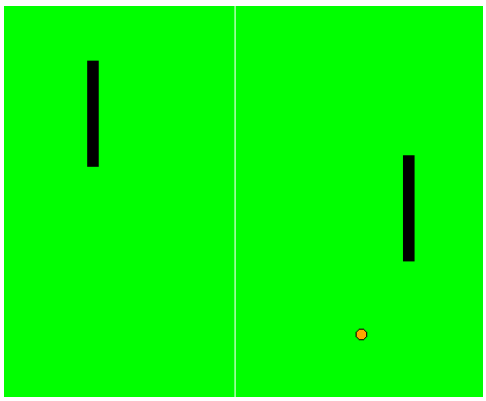

FIG. A.35 – Jeu de ping-pong

Le jeu est composé de 3 agents :

1. la palette de gauche ;
2. la palette de droite ;
3. la balle.

Pour chaque agent il faudra écrire une fonction de transition et lui attribuer un état.

Notre vision du jeu de ping-pong

Nous allons ici décrire la façon dont nous avons choisi de réaliser ce jeu. L'agent *balle* se déplace dans une certaine direction (vers la droite ou vers la gauche). Sa direction ainsi que sa position sont mémorisées à l'intérieur de son état. Pour visualiser le déplacement de

la balle, nous la faisons avancer à petits pas. Après chaque pas, la balle envoie sa position à la palette vers laquelle elle se dirige.

L'agent palette peut uniquement se déplacer de haut en bas à l'aide de certaines touches du clavier. Nous reviendrons plus tard sur ce point. L'état de palette contient la position de celle-ci. Lorsque la balle envoie sa position à la palette, cette dernière vérifie si la position (sur l'axe horizontal) de la balle est différente de sa propre position horizontale, qui correspond à la limite du terrain de jeu. Si la position est différente, cela veut dire que la balle n'a pas encore atteint la limite. La palette envoie donc un message à la balle pour lui dire de continuer d'avancer. Si les deux positions sont égales, cela signifie que la balle a atteint la limite du terrain. Dans ce cas il va falloir vérifier si la balle a bien touché la palette en comparant leurs positions verticales.

Si les positions verticales correspondent, la balle touche la palette et rebondit. La palette envoie alors un message à la balle pour lui dire de changer de direction (la nouvelle direction peut par exemple être un angle aléatoire).

Si les positions verticales ne correspondent pas, cela veut dire que la balle n'a pas touché la palette et qu'elle a donc quitté le jeu. C'est perdu et le jeu se termine.

Les états des agents

Voici le détail de l'état des agents palette et balle.

Etat de l'agent palette :

$$state(PosX\ PosY\ Tag)$$

Etat de l'agent balle :

$$state(PosX\ PosY\ Angle\ Tag)$$

$PosX$ est la position de l'agent sur l'axe X et $PosY$ est la position de l'agent sur l'axe Y. *Angle* est l'angle de la direction de la balle (convention trigonométrique) et Tag est la référence vers l'image de l'agent. C'est ici qu'apparaît une première difficulté car il faudra toujours veiller à ce que l'agent se trouve à la même position que son image (grâce à son tag).

Fonctions et procédures données

Fonctions de dessin

Voici deux fonctions mises à votre disposition afin de vous aider à dessiner à l'écran la balle et les palettes.

1. $TagBall = \{DrawBall\ X\ Y\}$ **:** cette fonction dessine à l'écran une balle à la position (X, Y) et renvoie un tag permettant d'identifier cette balle ;
2. $TagPaddle = \{DrawPaddle\ 200\ 200\}$ **:** cette fonction dessine à l'écran une palette de taille fixe (100 pixels) à la position (X, Y) qui correspond au haut de la palette,

et renvoie également un tag.

Procédure pour déplacer une image
Voici une procédure qui va vous permettre de déplacer une image :

$\{MoveTag\ Tag\ X\ Y\}$ **:** cette procédure déplace le dessin identifié par le tag de sa position horizontale actuelle posX vers posX + X et de sa position verticale actuelle posY vers posY + Y.
Attention ! La palette ne se déplace que de haut en bas donc seule la position Y sera utilisée (X sera donc toujours égal à 0).

Déplacement des palettes
Afin de rendre le jeu plus interactif, les palettes sont déplacées à l'aide de certaines touches du clavier. Il faut donc lier l'action d'enfoncer une touche au déplacement de l'agent palette, et également au déplacement de son image ! Voici quatre lignes de code à "copier/coller" dans votre application. Elles vous permettent de lier une touche à une action :

```
{Window bind(event:"<Up>" action:proc{$} {Send PaddleL move(~10)} end)}
{Window bind(event:"<Down>" action:proc{$} {Send PaddleL move(10)} end)}

{Window bind(event:"<F1>" action:proc{$} {Send PaddleR move(~10)} end)}
{Window bind(event:"<F2>" action:proc{$} {Send PaddleR move(10)} end)}
```

Cette ligne de code va lier (bind) à l'événement (event) "*appuyer sur une touche*" (par exemple : Down) une action $\{Send\ PaddleL\ move(10)\}$.
Dans le "Send" vous mettrez l'identificateur de votre agent palette à qui le message est destiné. Ce message permet à l'agent palette, lorsqu'il le reçoit, de changer de position et de déplacer son image. Vous pouvez bien entendu choisir le message que vous lui envoyez. Il faut prévoir deux touches pour chaque agent, une pour le déplacement vers le haut et l'autre pour le déplacement vers le bas.

Fonction Random
Une autre fonction fort intéressante dans le milieu des jeux vidéo est la fonction Random. Cette fonction renvoie un nombre aléatoire entier positif.

$$X\ =\ \{Random\}$$

Fonctions sinus, cosinus et conversion nombre entier nombre à virgule
Afin de gérer le déplacement de la balle dans toutes les directions, vous aurez besoin des fonctions qui calculent le sinus et cosinus d'un angle en degrés.

$$X\ =\ \{Cosinus\ 90\}$$
$$Y\ =\ \{Sinus\ 45\}$$

L'angle passé à la fonction est un nombre entier (Integer). Cette fonction renvoie un nombre à virgule (Float) car le sinus et le cosinus d'un angle sont compris entre -1 et 1. Etant donné que nous souhaitons travailler sur des nombres entiers, il faudra faire une conversion de nombre à virgule vers le nombre entier après avoir effectué vos calculs pour la direction de la balle.

$$UnNombreEntier \ = \ \{Float.toInt \ UnNombreAVirugle\}$$
$$UnNombreAVirgule \ = \ \{Int.toFloat \ UnNombreEntier\}$$

Exemple :

```
X = {Float.toInt 3.2}
{Send Browser X} % 3
Y = {Int.toFloat 3}
{Send Browser Y} % 3.0
```

Rappelons que l'on ne peut pas mélanger les deux types en Oz. On ne peut pas additionner un Float et un Integer !

Exemple de fonction de transition

Voici un exemple de fonction de transition pour l'agent Palette et l'agent Balle.

Fonction de transition de l'agent Palette de droite

```
fun{BrainPaddleR Msg state(PosX PosY Tag)}
   case Msg
      of start then
         {Send Ball moveLeft(138)}
         state(PosX PosY Tag)
      [] position(X Y Angle) then
         if(X>=PosX) then
            if(Y>=PosY andthen Y=<PosY+100) then
               NewAngle = ({Random} mod 90)+135 in
               {Send Ball moveLeft(NewAngle)}
            end
         else % quand on rate la balle
            if(Y =< YMIN orelse Y>=YMAX) then
               {Send Ball moveRight(~Angle)}
            else
               {Send Ball moveRight(Angle)}
            end
         end
         state(PosX PosY Tag)
      [] move(Y) then
         {MoveTag Tag 0 Y}
```

```
         state(PosX PosY+Y Tag)
      end
end
```

Rappel : pour créer une variable temporaire à l'intérieur d'une fonction ou procédure on utilise le mot clé *in* (cf cours2).

Fonction de transition de l'agent Balle

```
fun{BrainBall Msg state(PosX PosY Angle Tag)}
   case Msg
      of moveRight(A) then
         {MoveTag Tag {Float.toInt {Cosinus A}*5.0}
                      {Float.toInt {Sinus A}*5.0}}
         {Delay 50}
         {Send PaddleR position(PosX+{Float.toInt {Cosinus A}*5.0}
                                PosY+{Float.toInt {Sinus A}*5.0} A)}
         state(PosX+{Float.toInt {Cosinus A}*5.0}
               PosY+{Float.toInt {Sinus A}*5.0} A Tag)
      [] moveLeft(B) then
         {MoveTag Tag {Float.toInt {Cosinus B}*5.0}
                      {Float.toInt {Sinus B}*5.0}}
         {Delay 50}
         {Send PaddleL position(PosX+{Float.toInt {Cosinus B}*5.0}
                                PosY+{Float.toInt {Sinus B}*5.0} B)}
         state(PosX+{Float.toInt {Cosinus B}*5.0}
               PosY+{Float.toInt {Sinus B}*5.0} B Tag)
      end
end
```

Pour le déplacement de la balle nous nous sommes aidés du calcul trigonométrique à l'aide des fonctions sinus et cosinus des angles. Les fonctions sinus et cosinus renvoient un nombre à virgule, il faut donc utiliser des nombre à virgule lorsque l'on multiplie le résultat obtenu par une valeur. La valeur ici est 5.0. Cette valeur est choisie volontairement de manière à ce que la balle se déplace par pas de 5 pixels. On convertit le résultat en nombre entier de manière à garder dans l'état uniquement des nombres entiers.

A.7.3 Exercice

Nous vous demandons ici de réaliser la fonction de transition pour l'autre palette (celle de gauche) qui est, bien évidemment, fort similaire à celle de droite qui vous est donnée plus haut.
Nous vous demandons également de créer l'ensemble du programme (création des agents, création des images, ...).

Afin de rendre l'interface plus conviviale il existe dans le logiciel "Oz Turtle Graphics" un bouton "Change world" dans le menu "Edit". Choisissez "ping pong world" et le monde ressemblera à une table de ping pong.

Bon amusement ! ;-)

Annexe B

Dessins réalisés

B.1 Fougère

```
proc{Fougere L A O F1 F2}
   if(O==0) then skip
   else
   {Send Turtle left(A)}
   {Send Turtle forward(L)}
   {Fougere L*F1 A O-1 F1 F2}
   {Send Turtle penUp}
   {Send Turtle left(180)}
   {Send Turtle forward(L)}
   {Send Turtle left(180)}
   {Send Turtle right(2*A)}
   {Send Turtle penDown}
   {Send Turtle forward(L*F2)}
   {Fougere L*F1 A O-1 F1 F2}
   {Send Turtle penUp}
   {Send Turtle left(180)}
   {Send Turtle forward(L*F2)}
   {Send Turtle left(180)}
   {Send Turtle left(A)}
   {Send Turtle penDown}
   end
end

{Send Turtle left(90)}
{Send Turtle penDown}
{Send Turtle color(green)}
{Fougere 15 15 8 1 2}
```

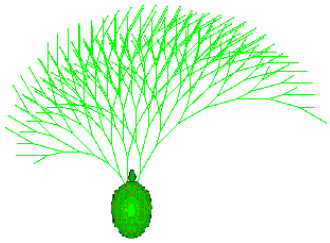

FIG. B.1 – Fougère

B.2 Flocon

```
proc{Khor L}
   if(L<5) then {Send Turtle forward(L)}
   else
      {Khor L div 3} {Send Turtle left(60)}
      {Khor L div 3} {Send Turtle right(120)}
      {Khor L div 3} {Send Turtle left(60)}
      {Khor L div 3}
   end
end
{Send Turtle penDown}
{Repeat 3 proc{$} {Khor 90} {Send Turtle right(120)} end}
```

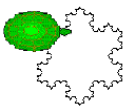

FIG. B.2 – Flocon

B.3 Puzzle

```
proc{Corner Size}
   {Send Turtle right(45)}
   {Send Turtle forward(Size)}
   {Send Turtle right(45)}
end

proc{OneSide Size Diag Level}
   if(Level == 0) then skip
   else
      {OneSide Size Diag Level-1}
```

```
      {Send Turtle right(45)} {Send Turtle forward(Diag)}  {Send Turtle right(45)}
      {OneSide Size Diag Level-1}
      {Send Turtle left(90)} {Send Turtle forward(Size)} {Send Turtle left(90)}
      {OneSide Size Diag Level-1}
      {Send Turtle right(45)} {Send Turtle forward(Diag)} {Send Turtle right(45)}
      {OneSide Size Diag Level-1}
   end
end

proc{Sierp Size Level}
   Diag = {Int.toFloat Size} / 2.0*{Float.sqrt 2.0} in
   {Repeat 4 proc{$} {OneSide Size {Float.toInt Diag} Level} {Corner {Float.toInt Diag}} e
end

{Send Turtle penDown}
{Sierp 5 4}
```

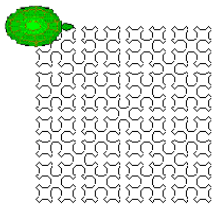

FIG. B.3 – Puzzle

B.4 Arbre

```
proc{Tree S}
   Size = {Float.toInt S} in
   if(Size < 5) then
      {Send Turtle forward(Size)}
      {Send Turtle left(180)}
      {Send Turtle penUp}
      {Send Turtle forward(Size)}
      {Send Turtle left(180)}
      {Send Turtle penDown}
      skip
   else
      {Send Turtle forward({Float.toInt S/3.0})}
      {Send Turtle left(30)}
```

```
      {Tree S*2.0/3.0} {Send Turtle right(30)}
      {Send Turtle forward({Float.toInt S/6.0})}
      {Send Turtle right(25)} {Tree S/2.0} {Send Turtle left(25)}
      {Send Turtle forward({Float.toInt S/3.0})}
      {Send Turtle right(25)} {Tree S/2.0} {Send Turtle left(25)}
      {Send Turtle forward({Float.toInt S/6.0})}
      {Send Turtle left(180)}
      {Send Turtle penUp}
      {Send Turtle forward(Size)}
      {Send Turtle penDown}
      {Send Turtle left(180)}
   end
end

{Send Turtle penDown}
{Send Turtle left(90)}
{Tree 50.0}
```

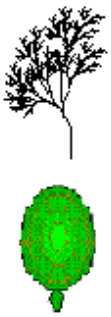

FIG. B.4 – Arbre

Annexe C

Comment utiliser le programme ?

Le programme est écrit en Oz. Assurez-vous que l'environnement Mozart (version 1.3.1) [13] est bien installé sur votre machine. Vous pouvez le télécharger à l'adresse suivante : **http ://www.mozart-oz.org**.

C.1 Compilation

Nous avons divisé notre application en quatre foncteurs :

1. *TurtleGraphics.oz* : le foncteur principal
2. *Metronome.oz*
3. *User.oz*
4. *ClassLinkedList.oz*

Avant de lancer le logiciel il faut compiler chaque foncteur séparément en tapant les lignes suivantes :

```
ozc -c Metronome.oz -o Metronome.ozf
ozc -c User.oz -o User.ozf
ozc -c ClassLinkedList.oz -o ClassLinkedList.ozf
ozc -c TurtleGraphics.oz -o TurtleGraphics.oza
```

C.2 Exécution

Pour lancer le logiciel, il suffit, après avoir compilé chaque foncteur, d'entrer la ligne de code suivante :

```
ozengine TurtleGraphics.oza
```

C.3 Manuel

Le chapitre 3 de cet ouvrage peut servir de manuel d'utilisation du logiciel.

Annexe D

Code de l'application

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          TurtleGraphics.oz                              %
%                  Mathieu Cuvelier & Isabelle Cambron                    %
%                                                                         %
%               functor principale : - GUI                                %
%                                    - Agent Turtle                       %
%                                    - Compiler                           %
%                                    - Environment                        %
%                                    - Worlds                             %
%                                                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

functor

import
   QTk at ´x-oz ://system/wp/QTk.ozf´
   Application(exit getArgs)
   System(show printInfo)
   Open(file)
   Narrator(´class´)
   ErrorListener(´class´)
   Compiler
   Inspector(inspect)
   OPIEnv(full)
   User
   Metronome
   ClassLinkedList

define
   %Taille et limite du monde de la tortue
   SIZE = 680.0 %960.0
   SIZEINT = {Float.toInt SIZE}
   XMIN = 20.0
   YMIN = 20.0
```

```
   XMAX = SIZE-20.0
   YMAX = SIZE-20.00

   %Variables globales
   PI = 3.141592653589793
   Canvas
   DELAY = {NewCell 5}
   StatusLabel
   LineLabel
   CharLabel
   TextHandle
   T %la TURTLE


%%%%%%%%%%%%%%% Choix du micromonde %%%%%%%%%%%%%%%%%

   C1 C2 C3 C4 C5
   Choice=td(radiobutton(text :"  first microworld"
                         init :true
                         return :C1
                         group :radio1)
             radiobutton(text :"second microworld"
                         return :C2
                         group :radio1)
             radiobutton(text :"  third microworld"
                         return :C3
                         group :radio1)
             radiobutton(text :"fourth microworld"
                         return :C4
                         group :radio1)
             radiobutton(text :"  fifth microworld"
                         return :C5
                         group :radio1))

   Button = button(text :"  Ok  "
                   justify :center foreground :yellow
                   action :toplevel#close
                   font :{QTk.newFont font(family :"Courrier" size :12)})

   Title = message(aspect :500
                   justify :center
                   font :{QTk.newFont font(family :"Courrier" size :16)}
                   text :"OZ TURTLE GRAPHICS\n" foreground :blue)

   Title2 = message(aspect :500
                    justify :center
                    font :{QTk.newFont font(family :"Courrier" size :12)}
                    foreground :red
                    text :"Welcome! Please choose a microworld\n")
```

```
   AboutMsg = td(borderwidth :20
                 Title
                 Title2
                 Choice
                 Button)

   ChoixMicro = {QTk.build td(title :"Choose Microworld" AboutMsg)}
   {ChoixMicro show} % affiche la fenetre du choix des micros
   {ChoixMicro set(action : proc{$} {System.show ´Goodbye...´} {Application.exit
0} end)}

   {Wait C5} %attend que l'utilisateur aie choisi un micromonde

   StateH
   StateM
   StateCW

   if(C1 == true) then %1er Microworld
      StateH = disabled
      StateM = disabled
      StateCW = disabled
   elseif(C2 == true) then %2e Microworld
      StateH = normal
      StateM = disabled
      StateCW = disabled
   elseif(C3 == true) then %3e Microworld
      StateH = normal
      StateM = disabled
      StateCW = disabled
   elseif(C4 == true orelse C5 == true) then % 4e et 5e Microworld
      StateH = normal
      StateM = normal
      StateCW = normal
   end


%%%%%%%%%%%%%%%    Le menu File    %%%%%%%%%%%%%%%%

   FileMenu = menu(command(text :"New" action :NewFile)
                   separator
                   command(text :"Save" action :SaveText)
                   command(text :"Open" action :proc{$} {LoadText StatusLabel}end)
                   separator
                   command(text :"Quit" action :Quit))


%%%%%%%%%%%%%%%     Le menu Help     %%%%%%%%%%%%%%%%
```

```
   HelpMenu = menu(command(text :"Instructions" action :Instructions)
                   separator
                   command(text :"About" action :About))


%%%%%%%%%%%%%%     Le menu Edit     %%%%%%%%%%%%%%%%%

   HandleTurtleWorld
   HandlePingPongWorld

   EditMenu = menu(cascade(text :"Delay"
                           menu :menu(tearoff :false
                                      radiobutton(
                                         text :"0 ms"
                                         action :proc{$} {ChangeDelay 0} end
                                         group :radiogroup
                                         init :false)
                                      radiobutton(
                                         text :"5 ms"
                                         action :proc{$} {ChangeDelay 5} end
                                         group :radiogroup
                                         init :false)
                                      radiobutton(
                                         text :"25 ms"
                                         action :proc{$} {ChangeDelay 25}
end
                                         group :radiogroup
                                         init :true)
                                      radiobutton(
                                         text :"100 ms"
                                         action :proc{$} {ChangeDelay 100}
end
                                         group :radiogroup
                                         init :false)
                                      radiobutton(
                                         text :"250 ms"
                                         action :proc{$} {ChangeDelay 250}
end
                                         group :radiogroup
                                         init :false)))
                   separator
                   cascade(text :"Change World Interface"
                           state :StateCW
                           menu :menu(tearoff :false
                                      radiobutton(
                                         text :"Turtle World"
                                         action :proc{$} {InitiateDrawing}
end
                                         group :radiogroup1
```

```
                                          init :true
                                          handle : HandleTurtleWorld)
                                       radiobutton(
                                          text :"Ping Pong World"
                                          action :proc{$} {DrawPingPongWorld}
end
                                          group :radiogroup1
                                          init :false
                                          handle : HandlePingPongWorld)
                                       radiobutton(
                                          text :"Time World"
                                          action :proc{$} {DrawTimeWorld} end
                                          group :radiogroup1
                                          init :false)))
                     separator
                     command(text :"Clear Text" action :ClearText)
                     command(text :"Clear Environment" action :ClearEnv))


%%%%%%%%%%%%%%%      Le menu Kill      %%%%%%%%%%%%%%%

   HalteMenu = menu(command(
                        text :"All Turtles"
                        action :proc{$}
                                  {ClassLinkedList.turtleLinkedList forall}
                                  {Halt}
                                  {User.browse ´All turtles have been killed
by user´}
                               end
                        state :StateH)
                     separator
                     command(
                        text :"All Metronomes"
                        action :proc{$}
                                  {Metronome.haltMetronome}
                                  {User.browse ´All metronomes have been killed
by user´}
                               end
                        state :StateM))



%%%%%%%%%%%%%%%      Les boutons       %%%%%%%%%%

   Buttons = lr(button(
                     text :"   Run   "
                     action :CompileText
                     padx :10
                     pady :5)
```

```
                button(
                   text :"  Clear World  "
                   action :
                      proc{$}
                         {User.browserObject clear}
                         B C in
                         {HandleTurtleWorld get(1 :B)}
                         {HandlePingPongWorld get(1 :C)}
                         if(B==true) then
                            {ClassLinkedList.turtleLinkedList forall}
                            {InitiateDrawing}
                         elseif(C==true) then
                            {DrawPingPongWorld}
                         else
                            {DrawTimeWorld}
                         end
                      end
                   padx :10 pady :5))


%%%%      Geometrie des composant de l'interface graphique     %%%%

   FileDesc = td(menubutton(glue :nw text :"File" menu :FileMenu))
   EditDesc = td(menubutton(glue :nw text :"Edit" menu :EditMenu))
   HalteDesc = td(menubutton(glue :nw text :"Kill" menu :HalteMenu))
   HelpDesc = td(menubutton(glue :nw text :"Help" menu :HelpMenu))

   Container=td(lr(FileDesc
                   EditDesc
                   HalteDesc
                   HelpDesc
                   glue :nw
                  )
                lr(
                   td(
                      text(handle :TextHandle
                           bg :white tdscrollbar :true
                           width :60%55
                           font :{QTk.newFont font(family :"Courier" size :13)}
                           glue :ns
                          )
                      label(handle :LineLabel glue :nw)
                    %pour afficher le numero de la ligne courante
                      label(handle :CharLabel glue :nw)
                    %pour afficher le numero du caractère courant
                      Buttons
                      label(handle :StatusLabel glue :nw)
                    %pour afficher le chemin du fichier
                      canvas(glue :we bg :black height :0.5)
```

```
                     User.browseDesc
                     glue :ns
                     )
                  td(
                     label(
                        font :{QTk.newFont font(family :"Helvetica" size :20)}
                        text :"LogOz : Oz Turtle Graphics" foreground :blue)
                     canvas(glue :ne
                            bg :grey
                            width :SIZE
                            height :SIZE
                            handle :Canvas))
                  )
               title :"Oz Turtle Graphics"
              )
   Window = {QTk.build Container}
   {Window show()}
   {Window set(action : Quit)}
% pour ajouter une action qd on click sur fermer la fenetre


%%%      Polling pour afficher le numero de la ligne courrante     %%%

   proc{BoucleAfficheLigne}
      Line = {TextHandle index(insert $)}.1
      Char = {TextHandle index(insert $)}.2 in
      {LineLabel set("")}
      {LineLabel set("Line : "#Line)}
      {CharLabel set("")}
      {CharLabel set("Char : "#Char)}
      {Delay 250} %Affichage du numero de ligne et du caractere 4 fois par
seconde
      {BoucleAfficheLigne}
   end

   thread
    %Assignation de Ctrl-C pour commencer à copier
      {TextHandle bind(event :"<Control-x>" action :CopyLocate)}
    %Assignation de Ctrl-S pour sauver
      {TextHandle bind(event :"<Control-s>" action :SaveText)}
      {BoucleAfficheLigne}
   end


%%%%%%%%%      Chargement des images      %%%%%%%%%%%%%

   TurtleImage0 = {QTk.newImage photo(file :"turtle0.gif")}
   TurtleImagep6= {QTk.newImage photo(file :"turtlep6.gif")}
   TurtleImagep4 = {QTk.newImage photo(file :"turtlep4.gif")}
```

```
   TurtleImagep3 = {QTk.newImage photo(file :"turtlep3.gif")}
   TurtleImagep2 = {QTk.newImage photo(file :"turtlep2.gif")}
   TurtleImage2p3 = {QTk.newImage photo(file :"turtle2p3.gif")}
   TurtleImage3p4 = {QTk.newImage photo(file :"turtle3p4.gif")}
   TurtleImage5p6 = {QTk.newImage photo(file :"turtle5p6.gif")}
   TurtleImagep = {QTk.newImage photo(file :"turtlep.gif")}
   TurtleImage7p6 = {QTk.newImage photo(file :"turtle7p6.gif")}
   TurtleImage5p4 = {QTk.newImage photo(file :"turtle5p4.gif")}
   TurtleImage4p3 = {QTk.newImage photo(file :"turtle4p3.gif")}
   TurtleImage3p2 = {QTk.newImage photo(file :"turtle3p2.gif")}
   TurtleImage5p3 = {QTk.newImage photo(file :"turtle5p3.gif")}
   TurtleImage7p4 = {QTk.newImage photo(file :"turtle7p4.gif")}
   TurtleImage11p6 = {QTk.newImage photo(file :"turtle11p6.gif")}


%%%%%%     Initialisation de l'environnement     %%%%%%%

   E = {New Compiler.engine init()}
   I = {New Compiler.interface init(E)}
   BaseEnv = {E enqueue(getEnv($))}


%%%%%%%     Inititialisation du monde (graphique)     %%%%%%%

   RectTag = {Canvas newTag($)}
   {Canvas create(rect XMIN YMIN XMAX YMAX
                  fill :white
                  outline :black
                  tags :RectTag)}


%%%%%%%%%%      Port Object de la tortue principale     %%%%%%%%%%%%%%%%%%%

   proc{NewPortObject Init Fun?P?Kill}
      proc{MsgLoop S State}
         case S of Msg|Sr then
            {MsgLoop Sr {Fun Msg State}}
         [] nil then skip
         end
      end
      Sin
   in
      thread
         local TId={Thread.this} in
            proc{Kill} %permet d'arreter la tortue a tout instant
               {Thread.terminate TId}
            end
         end
         {MsgLoop Sin Init}
```

```
      end
      {NewPort Sin P}
   end


%%%%    Conversion d'un angle en degres (Integer) vers un angle en radians
(Float)    %%%%

   fun{DegToRadian X}
      Deg = {Int.toFloat X}in
      Deg*((2.0*PI)/360.0)
   end


%%%%%    Conversion d'un radian (Float) en degres (Integer)    %%%%

   fun {RadianToDeg X}
      {Float.toInt X*(360.0/(2.0*PI))}
   end


%%%%% Fonction calculant la nouvelle direction vers laquelle la tortue est
orientee
% en fonction de son ancienne direction %%%%%%

   fun{DirectionCalc N Direction LR}
      case LR
      of left then
         NewAngle = {DegToRadian N}+Direction in
         if (NewAngle < 2.0*PI andthen NewAngle >= 0.0) then
            if (NewAngle >= 0.0 andthen NewAngle =< PI/16.0 orelse NewAngle
=< 2.0*PI andthen NewAngle >= 31.0*PI/16.0)
            then
               NewAngle#TurtleImage0
            elseif (NewAngle > PI/16.0 andthen NewAngle < PI/4.0) then
               NewAngle#TurtleImagep6
            elseif (NewAngle == PI/4.0) then
               NewAngle#TurtleImagep4
            elseif (NewAngle > PI/4.0 andthen NewAngle < 7.0*PI/16.0) then
               NewAngle#TurtleImagep3
            elseif (NewAngle >= 7.0*PI/16.0 andthen NewAngle =< 9.0*PI/16.0)
then
               NewAngle#TurtleImagep2
            elseif (NewAngle > 9.0*PI/16.0 andthen NewAngle < 3.0*PI/4.0)
then
               NewAngle#TurtleImage2p3
            elseif (NewAngle == 3.0*PI/4.0) then
               NewAngle#TurtleImage3p4
            elseif (NewAngle > 3.0*PI/4.0 andthen NewAngle < 15.0*PI/16.0)
```

```
then
               NewAngle#TurtleImage5p6
            elseif (NewAngle >= 15.0*PI/16.0 andthen NewAngle =< 17.0*PI/16.0)
then
               NewAngle#TurtleImagep
            elseif (NewAngle > 17.0*PI/16.0 andthen NewAngle < 5.0*PI/4.0)
then
               NewAngle#TurtleImage7p6
            elseif (NewAngle == 5.0*PI/4.0) then
               NewAngle#TurtleImage5p4
            elseif (NewAngle > 5.0*PI/4.0 andthen NewAngle < 23.0*PI/16.0)
then %2.0 3.0
               NewAngle#TurtleImage4p3
            elseif (NewAngle >= 23.0*PI/16.0 andthen NewAngle =< 25.0*PI/16.0)
then
               NewAngle#TurtleImage3p2
            elseif (NewAngle > 25.0*PI/16.0 andthen NewAngle < 7.0*PI/4.0)
then
               NewAngle#TurtleImage5p3
            elseif (NewAngle == 7.0*PI/4.0) then
               NewAngle#TurtleImage7p4
            elseif (NewAngle > 7.0*PI/4.0 andthen NewAngle < 31.0*PI/16.0)
then % /3.0
               NewAngle#TurtleImage11p6
            end
         elseif (NewAngle < 0.0) then
            {DirectionCalc N Direction+2.0*PI LR}
         elseif (NewAngle == 2.0*PI) then
            0.0#TurtleImage0
         else
            {DirectionCalc N Direction-2.0*PI LR}
         end
      [] right then
         {DirectionCalc {RadianToDeg (2.0*PI-{DegToRadian N})} Direction
left}
      end
   end


% Ajout de nouveaux elements (precedures, fonctions, variables, ...) a l'environnement
%

   EnvTemp = {NewCell enqueue(mergeEnv(env(´Turtle´ :T)))}
   proc {CompileText}
      {I reset()} % reset des messages d'erreur
      {E enqueue(mergeEnv(OPIEnv.full))}
      if(@EnvTemp \= nil) then
         {E @EnvTemp}
      end
```

```
      {E enqueue(mergeEnv(env(´Browse´ :User.browse)))}
      {E enqueue(mergeEnv(env(´Browser´ :User.viewer)))}

      %seulement a partir du 2e micromonde
      if(C2 == true orelse C3==true orelse C4==true orelse C5==true) then
         {E enqueue(mergeEnv(env(´Repeat´ :User.repeat)))}
      end

      if(C3 == true orelse C4==true orelse C5==true) then
      %seulement a partir du 3e micromonde
         {E enqueue(mergeEnv(env(´NewTurtle´ :NewTurtle)))}
         {E enqueue(mergeEnv(env(´Random´ :User.random)))}
      end

      if(C4==true orelse C5==true) then
         %seulement a partir du 4e micromonde
         {E enqueue(mergeEnv(env(´NewMetronome´ :Metronome.newMetronome)))}
      end

      if(C5 == true) then
         {E enqueue(mergeEnv(env(´NewAgent´ :User.newAgent)))}
         {E enqueue(mergeEnv(env(´DrawLine´ :DrawLine)))}
         {E enqueue(mergeEnv(env(´DeleteTag´ :User.deleteTag)))}
         {E enqueue(mergeEnv(env(´DrawBall´ :DrawBall)))}
         {E enqueue(mergeEnv(env(´MoveTag´ :User.moveTag)))}

         {E enqueue(mergeEnv(env(´DrawPaddle´ :DrawPaddle)))}
         {E enqueue(mergeEnv(env(´Window´ :Window)))}
      end
      {E enqueue(mergeEnv(env(´Cosine´ :User.cosinus)))}
      {E enqueue(mergeEnv(env(´Sine´ :User.sinus)))}
      {E enqueue(mergeEnv(env(´Tangent´ :User.tangente)))}

      {E enqueue(setSwitch(expression false))}
      {E enqueue(setSwitch(feedtoemulator true))}
      {StatusLabel set("Compiling ... ")}
      {E enqueue(feedVirtualString("declare "#{TextHandle get($)}))} % pour
ne pas devoir ercire declare
      {I sync()}

%%%%%%%%%% GESTION DES MESSAGES D'ERREUR %%%%%%%%%

      if {I hasErrors($)} then Ms Ligne Err in
         {I getMessages(?Ms)}
         try
            case Ms.1.kind
            of ´parse error´ then
               Ligne = {String.toAtom
```

```
                        {ByteString.toString
                         {ByteString.append {ByteString.make "error at line
"}
                          {ByteString.make Ms.1.items.1.2}}}
                       }
               Err = {String.toAtom Ms.1.msg}
            [] ´static analysis error´ then
               if(Ms.1.msg  == ´illegal arity in application´) then
                  Ligne = {String.toAtom
                           {ByteString.toString
                            {ByteString.append
                             {ByteString.make "error at line "}
                             {ByteString.make Ms.1.items.2.2.2.2.1.2}}}
                           }
                  Err = ´illegal arity in application´
               else
                  Ligne = {String.toAtom
                           {ByteString.toString
                            {ByteString.append {ByteString.make "error at
line "}
                             {ByteString.make Ms.1.items.2.1.2}}}
                           }
                  Err = Ms.1.msg
               end
            [] ´lexical error´ then
               {User.browse Ms}
               Ligne = ´error at line ´#Ms.1.items.1.2
            [] ´binding analysis error´ then
               Ligne = {String.toAtom
                        {ByteString.toString
                         {ByteString.append {ByteString.make "error at line
"}
                          {ByteString.make Ms.1.items.1.2}}}
                       }
               Err = Ms.1.msg
            [] ´illegal field selection´ then
               {User.browse ´illegal field selection´}
            else
               {User.browse Ms}
            end
            {User.browse Ligne}
            {User.browse Err}
         catch _ then
            {User.browse Ms}
            skip
         end
      else
         {System.show success}
      end
```

```
      thread {Delay 1000} {StatusLabel set("")} end
   end



%%%%%%%%%%%%%%%% TORTUE %%%%%%%%%%%%%%%%%%%%%%%%


   % fonction de transition de Turtle
   fun{TurtleBrain Msg state(pos(X Y) Direction Tag PenDown Color Name Kill)}
      case Msg
      of penDown then state(pos(X Y) Direction Tag true Color Name Kill)
      [] penUp then state(pos(X Y) Direction Tag false Color Name Kill)
      [] init then
         state(pos((SIZE-20.0)/2.0 (SIZE-20.0)/2.0) 0.0 Tag false Color Name
Kill)
      [] kill then
         {User.browse Name#´ has been killed by user´}
         {Tag delete}
         {Kill}
         state(pos(X Y) Direction Tag PenDown Color Name Kill)
      [] show then
         {User.browse state(Name {Float.toInt X} {Float.toInt Y} {RadianToDeg
Direction} PenDown Color)}
         state(pos(X Y) Direction Tag PenDown Color Name Kill)
      [] giveYourState(VariableToBind) then
         VariableToBind = state(Name {Float.toInt X} {Float.toInt Y} {RadianToDeg
Direction} PenDown Color)
         state(pos(X Y) Direction Tag PenDown Color Name Kill)
      [] name(N) then
         if({Atom.is N}) then
            state(pos(X Y) Direction Tag PenDown Color N Kill)
         else {User.browse ´name must be an atom : ´#N}
            state(pos(X Y) Direction Tag PenDown Color Name Kill)
         end
      [] left(N) then D in
         D={DirectionCalc N Direction left}
         {Tag delete}
         {Canvas create(image X Y image :D.2 tags :Tag)}
         {Delay @DELAY}
         state(pos(X Y) D.1 Tag PenDown Color Name Kill)
      [] right(N) then D in
         D={DirectionCalc N Direction right}
         {Tag delete}
         {Canvas create(image X Y image :D.2 tags :Tag)}
         {Delay @DELAY}
         state(pos(X Y) D.1 Tag PenDown Color Name Kill)
      [] forward(N) then
```

```
         NF = {Int.toFloat N} % distance à parcourir en float

         %% dépassement %%
         AB = NF*{Cos Direction}
         BC = NF*{Cos (PI/2.0 - Direction)} in

         %% depassement a DROITE
         if (XMAX-X < {Abs AB} andthen (Direction < PI/2.0 andthen Direction
>= 0.0 andthen XMAX-X =< Y-YMIN)
              orelse XMAX-X < AB andthen (Direction =< 2.0*PI andthen Direction
> 3.0*PI/2.0 andthen XMAX-X =< YMAX-Y)) then
            % depassementdroite
            Reste_x = AB - (XMAX-X)  %calcul du débordement selon l axe X
            Reste_pixel = NF - ( (AB - Reste_x) / {Cos Direction}) %calcul
du nombre de pixels qu'il reste à tracer
         in
               % depassement a DROITE et puis en HAUT
            if (Y-YMIN < {Abs AB} andthen Y-YMIN >= XMAX-X andthen Direction
> 0.0 andthen Direction < PI/2.0) then
               % depassementDroiteHaut
               Reste_y = Y-((NF-Reste_pixel)*{Sin Direction})-YMIN in
               {Tag move(XMIN+Reste_x-X YMAX-Reste_y-{Abs BC} + (NF-Reste_pixel)
* {Sin Direction}-Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                 {Float.toInt XMAX}
                                 {Float.toInt Y-((NF-Reste_pixel) * {Sin
Direction})})}
                  {Canvas create(line {Float.toInt XMIN} {Float.toInt YMAX-Reste_y}
                                 {Float.toInt XMIN+Reste_x}
                                 {Float.toInt YMAX-Reste_y-{Abs BC} + (NF-Reste_pixel)
* {Sin Direction}})}
               end
               state(pos(XMIN+Reste_x YMAX-Reste_y - {Abs BC} + (NF-Reste_pixel)
* {Sin Direction})
                     Direction Tag PenDown Color Name Kill)

            % depassement DROITE et puis en BAS
            elseif (YMAX-Y < {Abs BC} andthen YMAX-Y >= XMAX-X andthen Direction
> 3.0*PI/2.0 andthen Direction < 2.0*PI) then
               % depassementDroiteBas
               Reste_y = YMAX-Y+((NF-Reste_pixel)*{Sin Direction}) in
               {Tag move(XMIN+Reste_x-X YMIN+Reste_y+{Abs BC}+(NF-Reste_pixel)*{Sin
Direction}-Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y} {Float.toInt
XMAX}
                                 {Float.toInt Y-((NF-Reste_pixel)*{Sin Direction})})}
                  {Canvas create(line {Float.toInt XMIN} {Float.toInt YMIN+Reste_y}
```

```
                                {Float.toInt XMIN+Reste_x}
                                {Float.toInt YMIN + Reste_y + {Abs BC} +
(NF-Reste_pixel) * {Sin Direction}})}
               end
               state(pos(XMIN + Reste_x YMIN + Reste_y + {Abs BC} + (NF-Reste_pixel)
* {Sin Direction})
                     Direction Tag PenDown Color Name Kill)

            % depassement simple
            else
               {Tag move({Float.toInt (XMIN-X + Reste_x)} {Float.toInt (~NF
* {Sin Direction})})}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y} {Float.toInt
XMAX}
                                {Float.toInt Y - ((NF-Reste_pixel) * {Sin
Direction})}
                                fill :Color)}
                  {Canvas create(line {Float.toInt XMIN}
                                {Float.toInt Y - ((NF-Reste_pixel) * {Sin
Direction})}
                                {Float.toInt (XMIN + Reste_x)} {Float.toInt
Y-(NF * {Sin Direction})}
                                fill :Color)}
               end
               state(pos(XMIN + Reste_x Y-(NF * {Sin Direction})) Direction
Tag PenDown Color Name Kill)
            end

            %% dépassement à GAUCHE
         elseif (X-XMIN < {Abs AB} andthen Direction > PI/2.0 andthen Direction
=< PI andthen Y-YMIN >= X-XMIN
                  orelse X-XMIN < {Abs AB} andthen Direction >= PI andthen
Direction < 3.0*PI/2.0 andthen YMAX-Y >= X-XMIN) then
            Reste_x = {Abs AB}-(X-XMIN)
            Reste_pixel = NF - (({Abs AB}-Reste_x)/{Cos Direction})
         in

            % dépassement à GAUCHE puis en HAUT
            if (Y-YMIN < {Abs BC} andthen Y-YMIN >= X-XMIN andthen Direction
> PI/2.0 andthen Direction < PI) then
               Reste_y = Y - {Abs ((NF-Reste_pixel) * {Sin Direction})}-YMIN
in
               {Tag move(XMAX-Reste_x-X
                         YMAX-Reste_y-(NF * {Sin Direction}) - ((NF - Reste_pixel)
* {Sin Direction}) - Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                {Float.toInt XMIN}
```

```
                                    {Float.toInt (Y + ((NF - Reste_pixel) *
{Sin Direction}))}
                                    fill :Color)}
                  {Canvas create(line {Float.toInt XMAX} {Float.toInt YMAX-Reste_y}
                                    {Float.toInt XMAX - Reste_x}
                                    {Float.toInt YMAX - Reste_y - (NF * {Sin
Direction}) - ((NF-Reste_pixel) * {Sin Direction})}
                                    fill :Color)}
               end
               state(pos(XMAX-Reste_x YMAX-Reste_y - (NF * {Sin Direction})
- ((NF-Reste_pixel) * {Sin Direction}))
                     Direction Tag PenDown Color Name Kill)

            % dépassement à GAUCHE puis en BAS
            elseif (YMAX-Y < {Abs BC} andthen YMAX-Y >= X-XMIN andthen Direction
> PI andthen Direction < 3.0*PI/2.0) then
               Reste_y = YMAX - (Y + ((NF-Reste_pixel) * {Sin Direction}))
in
               {Tag move(XMAX-Reste_x - X YMIN+Reste_y + {Abs BC}-(NF-Reste_pixel)
* {Sin Direction} - Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y} {Float.toInt
XMIN}
                                    {Float.toInt (Y+((NF-Reste_pixel) * {Sin
Direction}))}
                                    fill :Color)}
                  {Canvas create(line {Float.toInt XMAX} {Float.toInt YMIN+Reste_y}
{Float.toInt XMAX-Reste_x}
                                    {Float.toInt YMIN + Reste_y + {Abs BC} -
(NF - Reste_pixel) * {Sin Direction}})}
               end
               state(pos(XMAX-Reste_x YMIN+Reste_y+{Abs BC}-(NF-Reste_pixel)*{Sin
Direction})
                     Direction Tag PenDown Color Name Kill)

            % depassement simple
            else
               {Tag move(XMAX-X-Reste_x (~NF*{Sin Direction}))}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y} {Float.toInt
XMIN}
                                    {Float.toInt (Y+((NF-Reste_pixel)*{Sin Direction}))}
                                    fill :Color)}
                  {Canvas create(line {Float.toInt XMAX}
                                    {Float.toInt (Y + ((NF - Reste_pixel) *
{Sin Direction}))}
                                    {Float.toInt XMAX-Reste_x}
                                    {Float.toInt Y - (NF*{Sin Direction})}
                                    fill :Color)}
```

```
               end
               state(pos(XMAX - Reste_x Y - (NF * {Sin Direction}))
                     Direction Tag PenDown Color Name Kill)
            end

            %% dépassement HAUT
         elseif (Y-YMIN < {Abs BC} andthen Direction > 0.0 andthen Direction
=< PI/2.0 andthen Y-YMIN =< XMAX-X
                  orelse Y-YMIN < {Abs BC} andthen Direction >=PI/2.0 andthen
Direction < PI andthen Y-YMIN =< X-XMIN) then
            Reste_y =  {Abs BC}-(Y-YMIN)
            Reste_pixel = NF - (({Abs BC}-Reste_y)/{Sin Direction})
         in

            % depassement HAUT et puis GAUCHE
            if (X-XMIN < {Abs AB} andthen X-XMIN > Y-YMIN andthen Direction
> PI/2.0 andthen Direction < PI) then
               Reste_x = X+(NF-Reste_pixel) * {Cos Direction}-XMIN in
               {Tag move(XMAX - Reste_x-{Abs AB} - (NF - Reste_pixel) * {Cos
Direction} - X
                         YMAX - Reste_y - Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                 {Float.toInt X + ((NF - Reste_pixel) * {Cos
Direction})}
                                 {Float.toInt YMIN}
                                 fill :Color)}
                  {Canvas create(line {Float.toInt XMAX-Reste_x} {Float.toInt
YMAX}
                                 {Float.toInt XMAX - Reste_x - {Abs AB} -
(NF-Reste_pixel) * {Cos Direction}}
                                 {Float.toInt YMAX-Reste_y}
                                 fill :Color)}
               end
               state(pos(XMAX - Reste_x - {Abs AB} - (NF-Reste_pixel) * {Cos
Direction} YMAX - Reste_y)
                     Direction Tag PenDown Color Name Kill)

            % depassement HAUT et puis DROITE
            elseif (XMAX-X < {Abs AB} andthen XMAX-X > Y-YMIN andthen Direction
> 0.0 andthen Direction < PI/2.0) then
               Reste_x = XMAX - (X + ((NF - Reste_pixel) * {Cos Direction}))
in
               {Tag move(XMIN + Reste_x + {Abs AB} - (NF - Reste_pixel) *
{Cos Direction} - X YMAX - Reste_y - Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X}
                                 {Float.toInt Y}
                                 {Float.toInt X + ((NF-Reste_pixel) * {Cos
```

```
Direction})}
                                {Float.toInt YMIN})}
                  {Canvas create(line {Float.toInt XMIN + Reste_x}
                                {Float.toInt YMAX}
                                {Float.toInt XMIN+Reste_x + {Abs AB} - (NF
- Reste_pixel) * {Cos Direction}}
                                {Float.toInt YMAX-Reste_y} fill :Color)}
               end
               state(pos(XMIN + Reste_x + {Abs AB} - (NF - Reste_pixel) *
{Cos Direction} YMAX - Reste_y)
                     Direction Tag PenDown Color Name Kill)

            % depassement haut simple
            else
               {Tag move(NF*{Cos Direction} YMAX-Y-Reste_y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X}
                                {Float.toInt Y}
                                {Float.toInt
                                 X + ((NF - Reste_pixel) * {Cos Direction})}
                                {Float.toInt YMIN}
                                fill :Color)}
                  {Canvas create(line {Float.toInt X + ((NF-Reste_pixel)
* {Cos Direction})}
                                {Float.toInt YMAX}
                                {Float.toInt X + (NF * {Cos Direction})}
                                {Float.toInt YMAX-Reste_y}
                                fill :Color)}
               end
               state(pos(X + (NF * {Cos Direction}) YMAX - Reste_y) Direction
Tag PenDown Color Name Kill)
            end

            %% dépassement BAS
         elseif (YMAX-Y < {Abs BC} andthen Direction > PI andthen Direction
=< 3.0*PI/2.0 andthen YMAX-Y =< X-XMIN
                 orelse YMAX-Y < {Abs BC} andthen Direction >= 3.0*PI/2.0
andthen Direction < 2.0*PI andthen YMAX-Y =< XMAX-X) then
            Reste_y = {Abs BC}-(YMAX-Y)
            Reste_pixel = NF - (({Abs BC}-Reste_y)/{Sin Direction})
         in

            % depassement BAS et puis GAUCHE
            if (X-XMIN < {Abs AB} andthen X-XMIN > YMAX-Y andthen Direction
> PI andthen Direction < 3.0*PI/2.0) then
               Reste_x = X-((NF-Reste_pixel)*{Cos Direction})-XMIN in
               {Tag move(XMAX-Reste_x-{Abs AB} + (NF-Reste_pixel) * {Cos
Direction}-X
                         YMIN+Reste_y-Y)}
```

```
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                 {Float.toInt X - ((NF-Reste_pixel) * {Cos
Direction})}
                                 {Float.toInt YMAX}
                                 fill :Color)}
                  {Canvas create(line {Float.toInt XMAX-Reste_x}
                                 {Float.toInt YMIN}
                                 {Float.toInt XMAX-Reste_x-{Abs AB} + (NF-Reste_pixel)
* {Cos Direction}}
                                 {Float.toInt YMIN+Reste_y}
                                 fill :Color)}
               end
               state(pos(XMAX-Reste_x-{Abs AB} + (NF-Reste_pixel) * {Cos
Direction} YMIN+Reste_y)
                     Direction Tag PenDown Color Name Kill)

            % depassement BAS et puis DROITE
            elseif (XMAX-X < {Abs AB} andthen Direction >= 3.0*PI/2.0 andthen
Direction < 2.0*PI andthen YMAX-Y =< XMAX-X) then
               Reste_x = XMAX-(X-((NF-Reste_pixel) * {Cos Direction})) in
               {Tag move(XMIN+Reste_x + {Abs AB} - {Abs (NF-Reste_pixel)
* {Cos Direction}}- X
                         YMIN+Reste_y-Y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                 {Float.toInt X-((NF-Reste_pixel) * {Cos
Direction})}
                                 {Float.toInt YMAX}
                                 fill :Color)}
                  {Canvas create(line {Float.toInt XMIN+Reste_x}
                                 {Float.toInt YMIN}
                                 {Float.toInt XMIN + Reste_x + {Abs AB} -
{Abs (NF-Reste_pixel) * {Cos Direction}}}
                                 {Float.toInt YMIN + Reste_y}
                                 fill :Color)}
               end
               state(pos(XMIN+Reste_x+{Abs AB} - {Abs (NF-Reste_pixel) *
{Cos Direction}}
                         YMIN+Reste_y)
                     Direction Tag PenDown Color Name Kill)

            % depassement bas simple
            else
               {Tag move(NF*{Cos Direction} YMIN-Y+Reste_y)}
               if (PenDown == true) then
                  {Canvas create(line {Float.toInt X} {Float.toInt Y}
                                 {Float.toInt X-((NF-Reste_pixel)*{Cos Direction})}
                                 {Float.toInt YMAX})}
```

```
                  {Canvas create(line {Float.toInt X-((NF-Reste_pixel)*{Cos
Direction})}
                                 {Float.toInt YMIN} {Float.toInt X+(NF*{Cos
Direction})}
                                 {Float.toInt (YMIN + Reste_y)})}
               end
               state(pos(X+(NF*{Cos Direction}) YMIN+Reste_y) Direction Tag
PenDown Color Name Kill)
            end

         % AUCUN dépassement
         else
            {Tag move({Float.toInt ((X+{Cos Direction}*{Int.toFloat N})-X)}
                      {Float.toInt ((Y-{Sin Direction}*{Int.toFloat N})-Y)})}
            if (PenDown == true) then
               {Canvas create(line {Float.toInt X} {Float.toInt Y}
                              {Float.toInt (X+{Cos Direction}*{Int.toFloat
N})}
                              {Float.toInt (Y-{Sin Direction}*{Int.toFloat
N})} fill :Color)}
            end
            state(pos(X + {Cos Direction} * {Int.toFloat N} Y - {Sin Direction}
* {Int.toFloat N})
                  Direction Tag PenDown Color Name Kill)
         end

      [] color(C) then
         if(C==red orelse C==black orelse C==green orelse C==blue orelse
C==yellow orelse C==pink
            orelse C==purple orelse C==orange orelse C==brown) then
            state(pos(X Y) Direction Tag PenDown C Name Kill)
         else {User.browse ´wrong color´}
            state(pos(X Y) Direction Tag PenDown Color Name Kill)
         end

      else
         {User.browse ´Turtle : unknown command ´#Msg}
         state(pos(X Y) Direction Tag PenDown Color Name Kill)
      end
   end


%%%%%%%%%     Active object Turtle     %%%%%%%%

   proc{Turtle Init Tid Kill}
      {NewPortObject Init TurtleBrain Tid Kill}
   end
```

```
%%%%%     Creation d'une nouvelle tortue    %%%%%

   proc{NewTurtle State ?UserTurtle}
      try
         TagImage = {Canvas newTag($)}
         NewState
         Position = {DirectionCalc State.4 0.0 left}
         Kill
      in
         {Canvas create(image State.2 State.3 image :Position.2 tags :TagImage)}
         {TagImage bind(event :"<1>"
                        action :proc{$} {Send UserTurtle show} end)}
         NewState = state(pos({Int.toFloat State.2} {Int.toFloat State.3})
                          Position.1 TagImage State.5 State.6 State.1 Kill)
         UserTurtle = {User.newPortObjectUser NewState TurtleBrain Kill}

         {ClassLinkedList.turtleLinkedList add(proc{$} {Kill} {TagImage delete}
end)}

      catch _ then {User.browse ´error : State incorrect´} skip % error
      end
   end


%%%%% initialisation de la tortue TURTLE %%%%%

   Tag = {Canvas newTag($)}
   {Canvas create(image ((SIZE-20.0)/2.0) ((SIZE-20.0)/2.0)
                  image :TurtleImage0
                  tags :Tag)}

   %Un click gauche sur la tortue va imprimer son etat dans le Browser embarque
   {Tag bind(event :"<1>" %<1> click gauche
             action :proc{$} {Send T show} end)}
   Kill
   {Turtle state(pos(((SIZE-20.0)/2.0) ((SIZE-20.0)/2.0)) 0.0 Tag false black
´turtle´ Kill) T Kill}


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   {ClassLinkedList.turtleLinkedList clear}
   {ClassLinkedList.turtleLinkedList add(proc{$} {Kill} {Tag delete} end)}


%%%%%%    Fonctions de dessin pour l'utilisateur    %%%%%

   % Dessine une ligne dans la GUI
   proc{DrawLine Length Orientation Color ?Res}
```

```
      try
         TagLine = {Canvas newTag($)} in %200
         {Canvas create(line
                        SIZEINT div 2
                        SIZEINT div 2
                        (SIZEINT div 2) + ({Float.toInt
                                            {Cos {DegToRadian Orientation}}
                                            * {Int.toFloat Length}})
                        (SIZEINT div 2) - ({Float.toInt
                                            {Sin {DegToRadian Orientation}}
                                            * {Int.toFloat Length}})
                        fill :Color tag :TagLine)}
         Res = TagLine
      catch _ then
         {User.browse ´Error in DrawLine´}
         skip
      end
   end


   % Dessine une pallette (ping-pong) dans la GUI
   proc{DrawPaddle X Y?Res}
      try
         TagLine = {Canvas newTag($)} in
         {Canvas create(rect X Y X+10 Y+100 fill :black tag :TagLine)}
         Res = TagLine
      catch _ then
         {User.browse ´Error in DrawPaddle´}
         skip
      end
   end


   % Dessine une balle dans la GUI
   proc{DrawBall X Y?Res}
      try
         TagBall = {Canvas newTag($)} in
         {Canvas create(oval X+10 Y+10 X Y fill :orange tag :TagBall)}
         Res = TagBall
      catch _ then
         {User.browse ´Error in DrawBall´}
         skip
      end
   end


%%%%%      NOTEPAD       %%%%%%%%%

   % Sauve le contenu du bloc note dans un fichier (.txt)
```

```
   proc{SaveText}
      Name={QTk.dialogbox save($)}
   in
      try
         File={New Open.file init(name :Name flags :[write create truncate])}
         Contents={TextHandle get($)}
      in
         {File write(vs :Contents)}
         {File close}
      catch _ then skip end
   end


   % Charge le contenu d'un fichier dans le bloc note
   proc{LoadText StatusLabel}
      Name={QTk.dialogbox load($)}
   in
      try
         {StatusLabel set("Opening file : "#Name)}
         File={New Open.file init(name :Name)}
         Contents={File read(list :$ size :all)}
         {Delay 750}
         {StatusLabel set("file : "#Name)}
      in
         {TextHandle set(Contents)}
         {File close}
      catch _ then
         {User.browse ´Error in LoadText´}
         skip
      end
   end


  % vide le bloc notes
   proc{ClearText}
      {TextHandle set("")}
   end


   %Détecte le curseur (position Src) et copie la ligne
   %1-Src à la nouvelle position du curseur
   %à la combinaison Ctrl-V
   DestChar = {NewCell nil}
   DestLin = {NewCell nil}
   proc{CopyLocate}

      SrcFirstLine
      SrcLastLine
      SrcFirstChar
```

```
      SrcLastChar in
      {User.browse ´Please select end of selection´}
      {User.browse ´and press Ctrl-c´}
      SrcFirstLine = {TextHandle index(insert $)}.1
      %récupère le numéro de ligne du curseur
      SrcFirstChar = {TextHandle index(insert $)}.2
      %récupère le numéro de caractère du curseur
      {TextHandle bind(
                     event :"<Control-c>"
                     action :proc{$}
                                SrcLastLine = {TextHandle index(insert $)}.1
                                SrcLastChar = {TextHandle index(insert $)}.2
                                {User.browse ´Please select the paste-target´}
                                {User.browse ´and press Ctrl-v´}
                             end)}
      {TextHandle bind(
                     event :"<Control-v>"
                     action :proc{$}
                                try
                                   DestLin := {TextHandle index(insert $)}.1
                                   DestChar := {TextHandle index(insert $)}.2
                                   {CopyLine SrcFirstLine SrcFirstChar SrcLastLine
SrcLastChar}% DestLine DestChar}
                                catch _ then
                                   {User.browse ´error in CopyLocate´}
                                   skip
                                end
                             end)
      }
   end

   %Copie la sélection SrcLine :SrcFirstChar-SrcLastChar
   %à la position DestLine :DestChar
   proc{CopyLine SrcFirstLine SrcFirstChar SrcLastLine SrcLastChar}% DestLine
DestChar}
      Selection in
      {TextHandle getText(coord(SrcFirstLine SrcFirstChar) coord(SrcLastLine
SrcLastChar) Selection)}
      {TextHandle insert(coord(@DestLin @DestChar) Selection)}
   end

%%%%     Vide tout ce qui a ete ajoute a l'environnement par l'utilisateur
%%%%

   proc{ClearEnv}
      {E enqueue(putEnv(BaseEnv))}
      {User.browse ´environment cleared´}
   end
```

```
%%%%%%%%%%     Arrete l'agent Turtle      %%%%%%%%%%

   proc{Halt}
      {E clearQueue()}
      {E interrupt()}
      {ClearEnv}
      {ClassLinkedList.turtleLinkedList clear}
      EnvTemp :=nil
   end

   proc{InitTurtle}
      NewT K in
      {Turtle state(pos(((SIZE-20.0)/2.0) ((SIZE-20.0)/2.0)) 0.0 Tag false
black ´turtle´ K) NewT K} %nouvelle tortue
      {ClassLinkedList.turtleLinkedList add(proc{$} {K} {Tag delete} end)}
      {Tag bind(event :"<1>"
                action :proc{$} {Send NewT show} end)}
      EnvTemp :=enqueue(mergeEnv(env(´Turtle´ :NewT)))
      {E enqueue(mergeEnv(env(´Turtle´ :NewT)))}
   end

%%%%%%%     Initialise le monde de la Tortue     %%%%%

   proc{InitiateDrawing}
      {Halt}
      try
         {Canvas create(rect XMIN YMIN XMAX YMAX fill :white outline :black
tags :RectTag)}
         {Canvas create(image ((SIZE-20.0)/2.0) ((SIZE-20.0)/2.0) image :TurtleImage0
tags :Tag)}
      catch _ then skip
      end
      {InitTurtle}

   end


%%%%     Initialise le monde du Ping-Pong     %%%%

   proc{DrawPingPongWorld}
      {Halt}
      {RectTag delete}
      {Tag delete}
      {Canvas create(rect XMIN YMIN XMAX YMAX fill :green outline :white
tags :RectTag)}
      {Canvas create(line XMAX/2.0 YMAX XMAX/2.0 YMIN fill :white)}
   end
```

```
%%%%%%     Initialise le monde de l'horloge     %%%%%%

   proc{DrawTimeWorld}
      {Halt}
      {RectTag delete}
      {Tag delete}
      {Canvas create(rect XMIN YMIN XMAX YMAX fill :white outline :black
tags :RectTag)}
      {Canvas create(text SIZEINT div 2 (SIZEINT div 2)-300 text :"12" fill :black)}
      {Canvas create(text SIZEINT div 2 (SIZEINT div 2)+300 text :"6" fill :black)}
      {Canvas create(text (SIZEINT div 2)-300 SIZEINT div 2 text :"9" fill :black)}
      {Canvas create(text (SIZEINT div 2)+300 SIZEINT div 2 text :"3" fill :black)}
      {Canvas create(text (SIZEINT div 2)+150 (SIZEINT div 2)-250 text :"1"
fill :black)}
      {Canvas create(text (SIZEINT div 2)-150 (SIZEINT div 2)-250 text :"11"
fill :black)}
      {Canvas create(text (SIZEINT div 2)-150 (SIZEINT div 2)+250 text :"7"
fill :black)}
      {Canvas create(text (SIZEINT div 2)+150 (SIZEINT div 2)+250 text :"5"
fill :black)}
      {Canvas create(text (SIZEINT div 2)+240 (SIZEINT div 2)-130 text :"2"
fill :black)}
      {Canvas create(text (SIZEINT div 2)-240 (SIZEINT div 2)+130 text :"8"
fill :black)}
      {Canvas create(text (SIZEINT div 2)-240 (SIZEINT div 2)-130 text :"10"
fill :black)}
      {Canvas create(text (SIZEINT div 2)+240 (SIZEINT div 2)+130 text :"4"
fill :black)}
   end


%%    Reinitialisation du monde    %%%

   proc {NewFile} % Vide l'environement, clear le texte, le Browser et le
dessin
      {TextHandle set("")}
      {User.browserObject clear}
      B C in
      {HandleTurtleWorld get(1 :B)}
      {HandlePingPongWorld get(1 :C)}
      if(B==true) then
         {ClassLinkedList.turtleLinkedList forall}
         {InitiateDrawing}
      elseif(C==true) then
         {DrawPingPongWorld}
      else
         {DrawTimeWorld}
      end
```

```
      {ClearEnv}
      {User.browserObject clear}
   end


%%%%%%%     Affiche le menu d'aide du logiciel     %%%%%

   proc {Instructions}
      InstTitle Inst InstButton InstMsg in
      InstTitle = message(aspect :3000 background :white
                          font :{QTk.newFont font(family :"CourrierHelvetica"
size :16 underline :true)}
                          text :"Instructions :" foreground :blue)

      Inst = message(aspect :3000 background :white
                     font :{QTk.newFont font(family :"Courrier" size :12)}
                     text :"\nKeyboard shortcuts :\n\n - Copy/Paste :\n    *
position the cursor at the begin of the text you want to copy and press ctrl-x\n
* position the cursor at the end of the text you want to copy and press ctrl-c\n
* position the cursor at the position where you want to copy and press ctrl-v\n\n
- Save text :\n    * ctrl-s\n\nBrowser Usage :\n\n - {Send Browser msgToBrowse} :
browses\" msgToBrowse\" in the Browser\n\nList of messages to the turtle :
{Send Turtle message}\n\n - forward(X) : asks the turtle to move X points
ahead\n - right(X) / left(X) (X = 0°,1°,..,360°) : asks the turtle to turn
to the right / left from X degrees\n - penUp / penDown : asks the turtle
to take her pen up / put her pen down\n - color(C) : changes the color of
the line (C = green, red, blue, yellow, black, purple, pink, orange)\n -
show : displays the state of the agent in the Browser\n - giveYourSate(X) :
binds the state of the agent to the variable X\n\nList of messages to a Metronome
agent : {Send Metronome message}\n\n  - start : starts the Metronome\n -
stop : stops the Metronome\n - register(Agent) : adds Agent to the list of
agents\n - message(Msg) : changes the message sent to the agents\n - clear :
clears the list and the message\n\nMathematical functions :\n\n  - Y = {Sine
X} : returns a float corresponding to the sine of X (= integer in degrees)\n
- Y = {Cosine X} : returns a float corresponding to the cosine of X (= integer
in degrees)\n - Y = {Tangent X} : returns a float corresponding to the tangent
of X (= integer in degrees)\n - Y = {Random} : returns a posivitve integer
random number\n\n Graphical tools :\n\n - Tag = {DrawLine Size Direction
Color} : draws a new line identified by\" Tag\"\n - {Delete Tag} : deletes
the image identified by\" Tag\"\n - {MoveTag Tag X Y} : moves the image (Tag)
from posX to posX + X and from posY to posY + Y\n - TagBall = {DrawBall X
Y} : draws a ball at the position (X,Y)\n - TagPaddle = {DrawPaddle X Y} :
draws a paddle of 100 pixels at the position (X,Y)\n\nHave fun!;-)")

      InstButton = button(text :"  Ok  "
                          justify :center
                          action :toplevel#close
                          font :{QTk.newFont font(family :"Courrier" size :10)})
```

```
      InstMsg = td(borderwidth :10
                   InstTitle
                   Inst
                   InstButton
                   background :white
                  )
      {{QTk.build td(title :"Instructions" InstMsg)} show}
   end


%%Change le delai entre chaque instruction execute par la tortue%%

   proc{ChangeDelay X}
      DELAY := X
      {User.browse ´delay changed´}
   end


%%%   Affiche les informations sur les auteurs  %%%%

   proc {About}
      AboutAuth
      AboutTitle
      AboutVersion
      AboutButton
      AboutMsg
   in
      AboutTitle = message(aspect :500 background :white
                           justify :center
                           font :{QTk.newFont font(family :"Courrier" size :16)}
                           text :"OZ TURTLE GRAPHICS\n" foreground :blue)

      AboutVersion = message(aspect :500 background :white
                             justify :center
                             font :{QTk.newFont font(family :"Courrier" size :12)}
                             text :"Version 1.1\n 20/05/2006\n" foreground :black)

      AboutAuth = message(aspect :500 background :white
                          justify :center
                          font :{QTk.newFont font(family :"Courrier" size :12)}
                          foreground :red
                          text :"Isabelle Cambron\n (isabelle.cambron@student.uclouvai
&\nMathieu Cuvelier\n(mathieu.cuvelier@student.uclouvain.be)\n")

      AboutButton = button(text :"  Ok  " background :white
                           justify :center foreground :yellow
                           action :toplevel#close
                           font :{QTk.newFont font(family :"Courrier" size :12)})
```

```
      AboutMsg = td(borderwidth :20
                    background :white
                    AboutTitle
                    AboutVersion
                    AboutAuth
                    AboutButton)
      {{QTk.build td(title :"About" AboutMsg)} show}
   end


%%%%%   Demande la confirmation pour quitter  %%%%

   proc {Quit}
      Msg
      ButtonOk
      ButtonKo
      QuitMsg
   in

      Msg = message(aspect :500
                    justify :center
                    font :{QTk.newFont font(family :"Courrier" size :14)}
                    foreground :red
                    text :"Are you sure?\n")

      ButtonOk = button(text :"  Yes  "
                        justify :center foreground :yellow
                        action :proc{$} {System.show ´Goodbye...´} {Application.exit
0} end
                        font :{QTk.newFont font(family :"Courrier" size :12)})

      ButtonKo = button(text :"  No  "
                        justify :center foreground :yellow
                        action :toplevel#close
                        font :{QTk.newFont font(family :"Courrier" size :12)})

      QuitMsg = td(borderwidth :20
                   Msg
                   lr(ButtonOk
                      ButtonKo)
                  )
      {{QTk.build td(title :"Quit" QuitMsg)} show}
   end
% fin du foncteur
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                User.oz                                      %
%                      Mathieu Cuvelier & Isabelle Cambron                    %
%                                                                             %
%   Fonctions et procedures utiliser Ã  l'exterieur par l'utilisateur    %
%                            Embeded Browser, Viewer                          %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

functor

import
   Browser
   OS

export
   Cosinus
   Sinus
   Tangente
   BrowseDesc
   Browse
   BrowserObject
   DeleteTag
   MoveTag
   Random
   NewPortObjectUser
   NewAgent
   Viewer
   Repeat

define
   PI = 3.141592653589793

%%%%%%      Port object utilise pour les agents crees par les utilisateurs
%%%%%%

   fun{NewPortObjectUser Init Fun?Kill}
      proc{MsgLoop Ss State}
         case Ss of Msg|Sr then
            {MsgLoop Sr {Fun Msg State}}
         [] nil then skip
         end
      end
      Sin
   in
      thread
         local TId={Thread.this} in
            proc{Kill} %permet d'arreter la tortue a tout instant
               {Thread.terminate TId}
```

```
            end
         end
         {MsgLoop Sin Init}
      end
      {NewPort Sin}
   end


   fun{NewPortObjectViewer Init Fun}
      proc{MsgLoop Ss State}
         case Ss of Msg|Sr then
            {MsgLoop Sr {Fun Msg State}}
         [] nil then skip
         end
      end
      Sin
   in
      thread
         {MsgLoop Sin Init}
      end
      {NewPort Sin}
   end
%%%%%%%%%%%%%%%%%%%%%%%%%      Browser embarque      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   Bhandle
   BrowseDesc = scrollframe(tdscrollbar :true lrscrollbar :true label(handle :Bhandle
width :1000 height :500))
   BrowserObject = {New Browser.´class´ init(origWindow :Bhandle)}

   proc{Browse X}
      {BrowserObject browse(X)}
   end

   {BrowserObject createWindow}
   {BrowserObject option(buffer size :100)}


%%%%%%%%%%%%%%%%%% AGENT AFFICHEUR %%%%%%%%%%%%%%%%%%%%

   fun{ViewerBrain X State}
      {Browse X}
      State
   end

   Viewer = {NewPortObjectViewer ´Browser´ ViewerBrain}


%%%%%%%%%%%%%%%% CREATION DE NOUVEAUX AGENTS %%%%%%%%%%%%%%%%
```

```
   proc{NewAgent Fun Memory ?UserAgent}
      try
         %Kill in
         %ajouter kill a la list des kill metronome
         UserAgent = {NewPortObjectUser Memory Fun _} %Kill
      catch _ then {Browse ´error in NewAgent parameters´} skip
      end
   end


%%%%%%%     la procedure repeat repete N fois la procedure P     %%%%%%%%

   proc{Repeat N P}
      try
         if N==0 then skip
         else
            {P}
            {Repeat N-1 P}
         end
      catch _ then {Browse ´error in Repeat´} skip
      end
   end


%%%    Conversion d'un angle en degres (Integer) vers un angle en radians
(Float)    %%%

   fun{DegToRadian X}
      Deg = {Int.toFloat X}in
      Deg*((2.0*PI)/360.0)
   end


%%%%%    Calcule le cosinus d'un angle en degres (Integer). Le resultat est
un Float    %%%%

   proc{Cosinus X ?Res}
      try
         Res = {Cos {DegToRadian X}}
      catch _ then
         {Browse ´Error in Cosine´}
         skip
      end
   end


%%%%%    Calcule le sinus d'un angle en degres (Integer). Le resultat est
un Float    %%%%
```

```
   proc{Sinus X ?Res}
      try
         Res = {Sin {DegToRadian X}}
      catch _ then
         {Browse ´Error in Sine´}
         skip
      end
   end


%%%%%    Calcule la tangente d'un angle en degres (Integer). Le resultat
est un Float    %%%%

   proc{Tangente X Res}
      try
         Res = {Tan {DegToRadian X}}
      catch _ then
         {Browse ´Error in Tangent´}
         skip
      end
   end


%%%%%%%%    Efface de la GUI le tag identifie par Tag    %%%%%%%%

   proc{DeleteTag Tag}
      try
         {Tag delete}
      catch _ then
         {Browse ´Error in DeleteTag´}
         skip
      end
   end


%%%%%%%%    Deplace sur la GUI le tag identifie par Tag    %%%%%%%%

   proc{MoveTag Tag X Y}
      try
         {Tag move(X Y)}
      catch _ then
         {Browse ´Error in MoveTag´}
         skip
      end
   end


%%%%%%%%    Renvoie un nombre aleatoire     %%%%%%%%
```

```
   {OS.srand 0}
   fun{Random}
      {OS.rand}
   end

end %functor
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                Metronome.oz                                 %
%                       Mathieu Cuvelier & Isabelle Cambron                   %
%                                                                             %
%       Ce functor contient tous les elements en rapport avec l'agent         %
%                                  Metronome                                  %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


functor

import
   User
   ClassLinkedList

export
   NewMetronome
   HaltMetronome

define

   {ClassLinkedList.metronomeLinkedList clear}
%%%%%%%%    TICKER : exactement 1 fois par seconde    %%%%%%%%

   fun{NewTicker}
      fun{Loop N}
         T={Time.time}
      in
         if T>N then {Delay 900}
         elseif T<N then {Delay 1100}
         else {Delay 1000} end
         N|{Loop N+1}
      end
   in
      thread {Loop {Time.time}} end
   end


%%%%%%%%    Concataine un element a une liste       %%%%%%%%

   fun{MyAppend Xs Y}
      case Xs
      of nil then [Y]
      [] X|Xr then Y|X|Xr
      end
   end
```

```
%%%%%%      fonction de transition de l'agent metronome     %%%%%%

   fun{MetronomeBrain Msg state(TurtleList MsgToSend OnOff Tid)}
      case Msg
      of start then
         if(@Tid == nil andthen TurtleList \= nil andthen MsgToSend \= nil)
then
            thread Tid :={Thread.this}
               for _ in {NewTicker} do
                  for T in TurtleList do {Send T MsgToSend} end
               end
            end
            state(TurtleList MsgToSend on Tid)
         else
            state(TurtleList MsgToSend off Tid)
         end
      [] register(Turtle) then NewList = {MyAppend TurtleList Turtle} in
         if(@Tid == nil) then
            state(NewList MsgToSend off Tid)
         else
            {KillThread @Tid}
            thread Tid :={Thread.this}
               for _ in {NewTicker} do
                  {ForAll NewList proc{$ O} {Send O MsgToSend} end}
               end
            end
            state(NewList MsgToSend on Tid)
         end
      [] clear then
         if(@Tid == nil) then
            state(nil nil off Tid)
         else
            {KillThread @Tid}
            state(nil nil off Tid)
         end
      [] message(M) then
         if(@Tid == nil) then
            state(TurtleList M off Tid)
         else
            {KillThread @Tid}
            thread Tid :={Thread.this}
               for _ in {NewTicker} do
                  {ForAll TurtleList proc{$ O} {Send O M} end}
               end
            end
            state(TurtleList M on Tid)
         end
      [] stop then
         {KillThread @Tid}
```

```
         Tid :=nil
         state(TurtleList MsgToSend off Tid)
      [] show then
         if(@Tid == nil) then
            {User.browse state(TurtleList MsgToSend)}
            {User.browse ´metronome is ´#off}
            state(TurtleList MsgToSend off Tid)
         else
            {User.browse state(TurtleList MsgToSend)}
            {User.browse ´metronome is ´#OnOff}
            state(TurtleList MsgToSend OnOff Tid)
         end
      else
         {User.browse ´Metronome : message not understand ´#Msg}
         state(TurtleList MsgToSend OnOff Tid)
      end
   end


%%%%%%%     Creer un nouveau Metronome     %%%%%%%
%A utiliser comme une fonction par l'utilisateur

   proc{NewMetronome State ?UserMetronome}
      try
         if({Tuple.is State} andthen {Width State}==2) then
            Tid = {NewCell nil}
            NewState = state(State.1 State.2 off Tid)
            Kill
         in
            UserMetronome = {User.newPortObjectUser NewState MetronomeBrain
Kill}
             {ClassLinkedList.metronomeLinkedList add(proc{$} {Send UserMetronome
stop} {Delay 10} {Kill} end)}
         else
            {User.browse ´Error : incorrect State´}
            skip %error
         end
      catch _ then
         {User.browse ´Error in NewMetronome´}
         skip
      end
   end


%%%%%%%     Arrete le metronome     %%%%%%%

   proc{HaltMetronome}
      {ClassLinkedList.metronomeLinkedList forall}
      {ClassLinkedList.metronomeLinkedList clear}
```

```
   end

   proc{KillThread Tid}
      {Thread.terminate Tid}
   end

end %functor
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              ClassLinkedList.oz                        %
%                      Mathieu Cuvelier & Isabelle Cambron               %
%                                                                        %
%       Ce functor contient l'implementation de la classe liste chaine  %
%       Cette liste chaine est utilisee pour memoriser les procedure    %
%       Kill permettant de tuer les port object creer                    %
%                                                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
functor

export

   TurtleLinkedList
   MetronomeLinkedList

define

   class LinkedList

      attr elem next

      meth init(elem :E<=null next :N<=null)
         elem :=E
         next :=N
      end

      meth add(E)
         next :={New LinkedList init(elem :E next :@next)}
      end

      meth forall
         if @elem \= null then
            try {@elem}
            catch _ then skip
            end
         end
         if @next \= null then {@next forall} end
      end

      meth clear
         if @elem \= null then elem :=null end
         if @next \= null then {@next clear} next :=null end
      end
   end

   TurtleLinkedList = {New LinkedList init(elem :null next :null)}
   MetronomeLinkedList = {New LinkedList init(elem :null next :null)}
end
```