

Stibbons

5 juin 2015

Julia Bassoumi - julia.bassoumi@etud.univ-montp2.fr
Florian Galinier - florian.galinier@etud.univ-montp2.fr
Adrien Plazas - adrien.plazas@etud.univ-montp2.fr
Clément Simon - clement.simon@etud.univ-montp2.fr

Encadrant : Michel Meynard

<https://gitlab.info-ufr.univ-montp2.fr/florian.galinier/stibbons.git>



Résumé

Ce projet vise à la création d'un langage de programmation multi-agents pour programmeurs débutants et avancés : le Stibbons. Nous l'avons réalisé en C++ et ses applications utilisent le framework Qt. Deux applications sont proposées pour répondre à deux cas d'utilisation différents : une application graphique permettant de développer des programmes Stibbons et de les voir s'exécuter directement, et une application en ligne de commande simplifiant l'exécution d'un programme et permettant un export régulier de données du modèle exécuté. Ce rapport expose le fonctionnement du langage Stibbons et de ses applications, ainsi que l'organisation que nous avons eu tout au long de la réalisation de ce projet.

Remerciements

Nous tenons tout particulièrement à remercier Michel Meynard pour avoir accepté de nous encadrer et pour son aide apportée tout au long du projet. Merci à lui de nous avoir accordé de son temps.

Nous souhaitons également remercier Jacques Ferber pour sa brillante introduction à la programmation multi-agents, ainsi que Sandrine Maton pour son aide apportée sur les méthodes agiles.

Le langage Stibbons tire son nom de la série de romans *Les Annales du Disque-Monde* de Terry Pratchett.

Table des matières

1	Introduction	5
1.1	Sujet	5
1.1.1	Objectif	5
1.1.2	Systèmes multi-agents	5
1.2	Cahier des charges	5
1.2.1	Fonctionnalités attendues	5
1.2.2	Contraintes	5
2	Analyse de l'existant	7
2.1	Logo	7
2.2	NetLogo	8
2.3	StarLogo	8
3	Analyse des outils	10
3.1	Outils de gestion de projet	10
3.1.1	Méthodes agiles	10
3.1.2	Git	12
3.1.3	Make	13
3.2	Tests unitaires	13
3.2.1	CppUnit	13
3.3	C++11	14
3.4	GDB	14
3.5	Outils d'analyse	14
3.5.1	Flex	14
3.5.2	Bison	15
3.6	Qt	17
3.6.1	Multi plateforme et multi langages	17
3.6.2	Modules	17
3.6.3	Concepts fondamentaux	17
3.6.4	Outils	18
3.6.5	Dessiner avec Qt	18
3.7	Latex	18
3.7.1	Généralités	18
3.7.2	Beamer	19
3.8	JSON Spirit	19
4	Modèle	20
4.1	Les classes d'agents	20
4.2	Les types	21

5	Interprète	25
5.1	Analyseur lexical	25
5.1.1	Jetons	25
5.1.2	Fonctionnement	26
5.2	Analyseur syntaxique	26
5.2.1	Arbre abstrait	27
5.2.2	Fonctionnement	28
5.3	Analyseur sémantique	28
5.3.1	Fonctionnement	28
5.3.2	Fonctionnalités	29
5.3.3	Organisation	29
6	Applications	31
6.1	Application graphique	31
6.1.1	Version 0.1	31
6.1.2	Version 0.2	31
6.1.3	Version 0.3	32
6.1.4	Version 0.4	33
6.1.5	Version 1.0	33
6.2	Application en ligne de commande	33
6.2.1	Utilisation	34
6.2.2	Fonctionnement	34
7	Conclusion	35
A	Documentation	39
A.1	Syntaxe	39
A.1.1	Flex	39
A.1.2	Bison	39
A.2	Propriétés standard	39
A.2.1	Attributs communs à tous les agents	39
A.2.2	Fonctions communes à tous les agents	40
A.2.3	Attributs du monde	41
A.2.4	Fonctions du monde	41
A.2.5	Attributs des tortues	41
A.2.6	Fonctions des tortues	41
A.2.7	Attributs des zones	42
B	Tutoriel	43
B.1	Tutoriel	43
B.1.1	Salut, monde!	43
B.1.2	Les premiers agents	43
B.1.3	Dessiner un carré	43
B.1.4	Répéter	44
B.1.5	Boucler	44
B.1.6	Agents typés	44
B.1.7	Fonctions	44
B.1.8	Couleurs	45
B.1.9	Propriétés d'autres agents et parent	45
B.1.10	Le monde	46
B.1.11	Les zones	46
B.1.12	Directives de monde	47

B.1.13 Messages	47
C Résumés des réunions	49
C.1 27 janvier 2015	49
C.2 3 février 2015	49
C.3 10 février 2015	50
C.3.1 Analyse de l'existant	50
C.3.2 Analyse des outils	50
C.4 24 février 2015	50
C.5 16 mars 2015	51
C.6 17 mars 2015	51
C.7 7 avril 2015	52
D Listing	53
D.1 Flex	53
D.2 Bison	58
D.3 CppUnit	73
D.4 Json Spirit	75
Bibliographie	78

Chapitre 1

Introduction

1.1 Sujet

1.1.1 Objectif

Le projet Stibbons vise à créer un interprète d'un dérivé de Logo, un langage de programmation permettant l'animation d'un agent mobile, dit « tortue » (cf. 2.1). Nous avons choisi de développer un langage multi-agents, à l'instar de NetLogo ou StarLogo (cf. 2.2 et 2.3), rendant ainsi l'objectif de ce projet double : d'une part la réalisation d'un interprète capable d'analyser du code écrit dans un dérivé de Logo, d'autre part la réalisation d'une application graphique capable de représenter l'évolution des agents.

1.1.2 Systèmes multi-agents

Les systèmes multi-agents sont une approche de l'intelligence artificielle visant à faciliter la résolution d'un problème en le découplant en plusieurs sous-problèmes. Ainsi, plutôt que de chercher à développer une intelligence unique complexe capable de résoudre le problème, l'approche multi-agents vise plutôt à créer des multitudes d'intelligences capables de résoudre une petite partie du problème, et de compter sur l'intelligence collective émergente pour voir apparaître la solution au problème [Ferber, 1995].

Ainsi, on peut prendre en exemple les fourmis qui, bien que n'ayant individuellement qu'une capacité limitée, sont capables de survivre grâce à la synergie de leurs colonies.

1.2 Cahier des charges

1.2.1 Fonctionnalités attendues

Bien que la méthode de développement utilisée ait fait apparaître de nombreuses fonctionnalités souhaitables au fur et à mesure du projet, un certain nombre d'entre elles nous ont semblé indispensables dès le début :

- chaque agent devait être capable de communiquer avec les autres agents ;
- chaque agent devait pouvoir modifier d'une certaine façon le monde ;
- le langage devait comprendre des structures des langages de programmation impérative modernes, telles que des conditionnelles, des boucles, des fonctions, etc. ;
- l'utilisateur devait pouvoir voir le monde où évoluent ces agents.

1.2.2 Contraintes

Nous avons dès le début isolé un certain nombre de contraintes que nous souhaitons pour ce projet :

- l'utilisation d'un langage non interprété, de type C ou C++, pour son développement ;
- chaque agent devait évoluer parallèlement aux autres agents ;
- l'utilisation d'une méthode agile pour développer le projet.

Chapitre 2

Analyse de l'existant

2.1 Logo

Le Logo est un langage de programmation apparu dans les années 60 dont l'objectif était alors de permettre à des personnes possédant peu de connaissances en informatique et en programmation (des enfants par exemple) de découvrir ce domaine de manière ludique et interactive. Le langage permettait de contrôler une tortue (un robot) capable d'avancer, de tourner, et d'abaisser un crayon ou un feutre pour dessiner sur une feuille placée au sol. Les instructions entrées permettaient ainsi de tracer des formes, permettant une représentation très visuelle du code (la tortue physique est remplacée dans les implémentations modernes par une tortue virtuelle).

Le langage Logo en lui-même est un dérivé du Lisp (il est d'ailleurs parfois nommé « Lisp sans parenthèses ») et possède deux types de données : les mots (chaînes de caractères) et les listes.

Du fait du public visé, les instructions de base (par exemple **forward**, **left**, **pendown**, etc.) et les structures du type procédures, boucles ou conditionnelles sont écrites de façon à être clairement explicites (cf. 2.1, 2.2 et 2.3). Cependant, comme expliqué dans l'article [Pea, 1987], des études sur Logo ont montré que les enfants, hormis quelques exceptions, n'arrivent pas à créer un programme entier et codent « ligne par ligne » ce qui les empêche de créer un modèle complexe et de cerner l'ensemble de la syntaxe de Logo.

```
to <nom de la procédure> :<paramètre>
  <instructions>
  output <valeur à retourner>
end
```

Listing 2.1 – Procédure en Logo

```
repeat <nb fois> [liste d'instructions]
```

Listing 2.2 – Boucle en Logo

```
if <test> [liste d'instructions si vrai]
ifelse <test> [liste d'instructions si vrai] [liste d'instructions si faux]
```

Listing 2.3 – Conditionnelles en Logo

Les instructions amènent la tortue à se déplacer suivant une certaine distance et un certain angle. On l'oriente ainsi suivant des coordonnées polaires.

2.2 NetLogo

NetLogo est un environnement de modélisation programmable développé à la Northwestern University (ref. [Wilensky, 1999]) et permettant de simuler et d’observer des phénomènes naturels et sociaux au fil du temps. Il permet de donner des instructions à des agents et d’observer leur évolution et les connexions inter-agents au niveau micro (individu par individu) comme au niveau macro (monde global). Il est aussi utilisé dans de nombreux domaines de recherche comme l’économie, la biologie, la physique, la chimie, etc. et de nombreux articles ont été publiés à son sujet.

Au niveau du programme en lui-même, NetLogo est composé de 3 onglets :

- l’onglet info : la documentation du code ;
- l’onglet code : le code permettant de créer le modèle du monde ainsi que le comportement des agents y sont implémentés ;
- l’onglet interface :
 - la partie « observation » qui est représentée par une fenêtre où l’on verra notre modèle évoluer dans le temps (le rendu pouvant être effectué en 2D comme en 3D) ;
 - la partie « construction » où l’on peut ajouter des widgets interagissant avec le code.

Les widgets de la partie construction permettent d’interagir avec des procédures ou des variables du code. Par exemple, l’ajout d’un slider « nb_population » pourrait permettre de définir le nombre de tortues à créer sans modifier le code. On pourrait également y faire apparaître des boutons pour démarrer ou arrêter des procédures, des graphes pour observer des variations, des interrupteurs pour gérer des variables globales, des notes, etc.

On peut également contrôler le temps, ralentir pour mieux observer, accélérer pour voir ce que produit le modèle.

NetLogo permet donc une interaction très rapide entre le code et le rendu graphique, permettant au développeur de « jouer » avec le modèle en modifiant facilement certaines conditions et donc d’ajuster immédiatement le code comme il le souhaite.

Une riche documentation et de nombreux tutoriels sont fournis sur le site officiel du langage, ce qui permet une prise en main simple et ludique.

NetLogo est un logiciel libre et open source, sous licence GPL. Il fonctionne sur la machine virtuelle Java, et est donc opérable sur de nombreuses plateformes (Mac, Windows, Linux, etc.). D’après son site officiel, NetLogo est décrit comme la prochaine génération de langages de modélisation multi-agents, tout comme StarLogo.

2.3 StarLogo

Tout comme NetLogo, StarLogo est un environnement de modélisation programmable permettant de simuler et d’observer des phénomènes naturels et sociaux au fil du temps. Ils ont les mêmes objectifs d’étude, d’éducation et de « programmation facile » ainsi qu’un aperçu direct du rendu (ref. [Resnick et al., 2008]).

Là où StarLogo diffère est qu’il n’est pas nécessaire de connaître une seule ligne de code. En effet, StarLogo se base sur un principe de bouton ; pour une partie de code donnée, un bouton y correspond. Les boutons peuvent être liés entre eux, permettant de créer des actions plus complexes. Tout bouton peut être positionné dans une page, qui correspond aux différents « environnements » du modèle : le monde, un certain type de tortues, les patches, etc. Par exemple, pour créer douze tortues lors d’une initialisation globale du modèle, il faut se positionner sur la page `setup`, y placer le bouton `setup` attaché du bouton `create Turtles - num`, puis modifier le `num` en 12.

StarLogo est composé de deux fenêtres :

- la fenêtre code : c’est ici que les boutons sont placés dans les différentes pages pour générer le code ;

— la fenêtre vue : on y aperçoit les résultats du code généré, le rendu pouvant être en 2D comme en 3D.

Au niveau historique, StarLogo avait d’abord été créé pour Mac lors de la première version, puis après plusieurs années, une version pour tout type d’environnement a vu le jour et la version uniquement pour Mac fut rebaptisée MacStarLogo. Plusieurs versions sont apparues mais c’est la version 2.1 de 2004 qui reste la plus récente.

StarLogo est disponible sous la même licence et pour le même environnement d’exécution que son confrère NetLogo : c’est un logiciel libre sous licence GPL qui fonctionne sur la machine virtuelle Java, d’où son gain de portabilité après la version MacStarLogo.

Chapitre 3

Analyse des outils

3.1 Outils de gestion de projet

3.1.1 Méthodes agiles

Pour réaliser ce projet, nous avons choisi d'utiliser la méthode agile SCRUM. Nous nous sommes principalement servi des cours de gestion de projet de ce semestre, en particulier des interventions de Sandrine Maton. Utiliser une méthode agile permet d'avoir un rendu fonctionnel à chaque fin de sprint, et donc de s'assurer d'avoir un rendu final. De plus, ces méthodes étant de plus en plus populaires dans le milieu des entreprises, nous souhaitons donc l'expérimenter.

La méthode Scrum est une méthode itérative, dont chaque itération est nommée « sprint ». Un sprint se décompose en trois parties : une réunion initiale, où les objectifs du sprint sont définis et les tâches réparties, le sprint en lui-même, durant lequel le développement a lieu, et une réunion pour faire le point sur le sprint écoulé.

Backlog initial

Le sprint 0 correspond à la première période du projet, durant laquelle nous avons fait notre analyse de l'existant et des outils. C'est également durant ce sprint que nous avons choisi les différentes fonctionnalités que nous voulions voir dans le projet. Pour chacune d'entre elles, nous écrivons un scénario utilisateur, une courte description de ce que l'utilisateur doit pouvoir faire via une fonctionnalité.

Une fois ces fonctionnalités définies, nous avons priorisé celles-ci et établi le « backlog » : la feuille de route de notre projet (cf. Table 3.1).

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
1	L'utilisateur écrit du code dans un éditeur	200					
2	L'utilisateur importe du code dans le logiciel	1100	Importer du code Stibbons depuis un fichier, vérifier que le code obtenu est bien identique à celui du fichier.	4h	1	Fini	1h
3	L'utilisateur visualise les rapports d'erreurs du code	400					
4	L'utilisateur visualise l'évolution du modèle	1400	Vérifier que l'interprétation d'instructions données fait bien évoluer comme prévu la tortue dans son environnement.	24h	1	Fini	70h
5	L'utilisateur modifie la vitesse (pause, pas à pas, parallèle)	700					
6	Le « dieu-tortue » interprète le code de l'utilisateur	1500	Lancer l'interprétation pour : repeat 4 fd 1 rt 90 (suivant syntaxe) ainsi que pour du code avec des erreurs : repeat 4 ... par exemple ou repeat 4 fd 1 rt	12h	1	Fini	32h

TABLE 3.1: Le backlog au début du sprint 2

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
7	L'utilisateur crée une nouvelle tortue (avec un code)	600	Lancer l'interprétation pour : create-turtle et observer une nouvelle tortue apparaître dans l'interface graphique.	7,5h	2		
8	Les tortues s'exécutent en parallèles	1200	Lancer l'interprétation pour deux tortues d'un bout de code et observer l'exécution parallèle via des écritures dans le terminal (Je suis tortue 1 et Je suis tortue 2 par exemple)	42,5h	2		
9	Les tortues communiquent directement entre elles	900					
10	Une tortue se déplace dans l'environnement	1300	Ecriture d'instructions simples : repeat 4 fd 1 rt 90 (suivant syntaxe) - Renvoi de la position de la tortue après chaque déplacement : where_am_i(); (suivant syntaxe)	16h	1	Fini	24h
11	Les tortues communiquent avec les zones de l'environnement	1000					
12	L'utilisateur exporte le code	500					
13	L'utilisateur exporte le modèle	300					
14	L'utilisateur ajoute une entrée	100					
15	L'utilisateur remet à zéro l'environnement	800					
16	L'utilisateur utilise des variables dans le code	1275	Ecrire a = 90 fd a et observer la tortue qui avance.	12h	2		
17	L'utilisateur définit des fonctions personnalisées dans le code	1250	Ecrire fonction f () fd 90 f () et observer la tortue qui avance.	23,5h	2		
18	Les tortues communiquent via l'environnement	950					
19	L'utilisateur utilise des conditionnelles	550	Ecrire if(false) fd 90 et observer que la tortue ne bouge pas. Refaire le même test avec if(true) et observer que la tortue bouge.	3,5h	2		
20	L'utilisateur utilise des boucles	575	Ecrire pd repeat 4 fd 40 rt 90 et observer que la tortue dessine un carré.	7h	2		

TABLE 3.1: Le backlog au début du sprint 2

Lors de la réunion de début de sprint, les fonctionnalités à ajouter sont choisies et pour chacune d'entre elles une description du test à effectuer afin de la valider était ajoutée au backlog. Nous faisons également une estimation du nombre d'heures nécessaires à chacune des nouveautés. À la fin d'un sprint, nous estimons le temps passé sur chaque tâche et nous faisons le point sur notre avancée dans le projet.

Backlog final

Sur le backlog final (cf. Table 3.2), nous pouvons voir les ajouts effectués, le changement de statut des fonctionnalités réalisées, et le temps réel passé à les rendre utilisables.

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
1	L'utilisateur écrit du code dans un éditeur	200	L'utilisateur écrit du code dans un éditeur intégré	20h	5	Fini	19h
2	L'utilisateur importe du code dans le logiciel	1100	Importer du code Stibbons depuis un fichier, vérifier que le code obtenu est bien identique à celui du fichier.	4h	1	Fini	1h
3	L'utilisateur visualise les rapports d'erreurs du code	400	Exécuter pd 50 et constater une erreur.	8h	4	Fini	8h
4	L'utilisateur visualise l'évolution du modèle	1400	Vérifier que l'interprétation d'instructions données fait bien évoluer comme prévu la tortue dans son environnement.	24h	1	Fini	70h
5	L'utilisateur modifie la vitesse (pause, pas à pas, parallèle)	700	Faire varier la vitesse et observer le changement dans l'exécution des tortues.	12h	4	Fini	14h

TABLE 3.2: Le backlog à la fin du sprint 5

id	Scénario utilisateur	Priorité	Tests	Estimation	Sprint	Statut	Temps réel
6	Le « dieu-tortue » interprète le code de l'utilisateur	1500	Lancer l'interprétation pour : repeat 4 fd 1 rt 90 (suivant syntaxe) ainsi que pour du code avec des erreurs : repeat 4 ... par exemple, ou repeat 4 fd 1 rt	12h	1	Fin	32h
7	L'utilisateur crée une nouvelle tortue (avec un code)	600	Lancer l'interprétation pour : create-turtle et observer une nouvelle tortue apparaître dans l'interface graphique.	7,5h	2	Fin	4h
8	Les tortues s'exécutent en parallèles	1200	Lancer l'interprétation pour deux tortues d'un bout de code et observer l'exécution parallèle via des écritures dans le terminal (Je suis tortue 1 et Je suis tortue 2 par exemple)	42,5h	2	Fin	40h
9	Les tortues communiquent directement entre elles	900	Ecrire send(t,"Je suis là") avec t une tortue qui a pour code : m = recv() if (m == "Je suis là") fd 50 et observer la tortue avancer	30h	3	Fin	16h
10	Une tortue se déplace dans l'environnement	1300	Ecriture d'instructions simples : repeat 4 fd 1 rt 90 (suivant syntaxe) - Renvoi de la position de la tortue après chaque déplacement : where_am_i(); (suivant syntaxe)	16h	1	Fin	24h
11	Les tortues communiquent avec les zones de l'environnement	1000	Ecrire if(zone.color == red) color = blue sur une zone de couleur rouge et observer la tortue changer de couleur.	30h	3	Fin	12h
12	L'utilisateur exporte le code	500	L'utilisateur sauvegarde son code dans un fichier externe	1h	5	Fin	1h
13	L'utilisateur exporte le modèle	300	Sauvegarder le modèle et observer les données en sortie (JSON)	30h	4	Fin	22h
14	L'utilisateur ajoute une entrée	100					
15	L'utilisateur remet à zéro l'environnement	800	Remettre à zéro après une exécution et observer que le monde est vierge.	20h	4	Fin	23h
16	L'utilisateur utilise des variables dans le code	1275	Ecrire a = 90 fd a et observer la tortue qui avance.	12h	2	Fin	3h
17	L'utilisateur définit des fonctions personnalisées dans le code	1250	Ecrire function f () fd 90 f () et observer la tortue qui avance.	23,5h	2	Fin	11h
18	Les tortues communiquent via l'environnement	950	Ecrire broadcast(20,"Je suis là") et dans une autre tortue dans le rayon avec le code m = recv() if (m == "Je suis là") fd 50 et observer la tortue avancer.	10h	3	Fin	2h
19	L'utilisateur utilise des conditionnelles	550	Ecrire if(false) fd 90 et observer que la tortue ne bouge pas. Refaire le même test avec if(true) et observer que la tortue bouge.	3,5h	2	Fin	3,5h
20	L'utilisateur utilise des boucles	575	Ecrire pd repeat 4 fd 40 rt 90 et observer que la tortue dessine un carré.	7h	2	Fin	3,5h
21	L'utilisateur définit des fonctions avec paramètres	1225	Ecrire function f(a) fd a f(90) et observer la tortue avancer.	10h	3	Fin	16h
22	L'utilisateur instancie des agents avec paramètres	560	Ecrire agent wolf (a) fd a new wolf (50) et observer la nouvelle tortue avancer.	10h	3	Fin	16h
23	L'utilisateur modifie la couleur d'un agent (tortue ou zone)	450	Ecrire new agent color = red et observer la nouvelle tortue rouge.	3h	3	Fin	2h
24	La tortue accède aux données de son parent	50	Taper parent.color= blue et observer le parent devenir bleu	1h	5	Fin	1h
25	L'utilisateur peut stocker des valeurs dans une table	540	On stocke le retour dans t= {1,2}	10h	4	Fin	9h
26	L'utilisateur parcourt un tableau	530	Faire foreach(f : {1,2}) println(f)	4h	5	Fin	4h
27	L'utilisateur lance l'application sans interface graphique	250	Lancer l'application et regarder les fichiers d'exportations	16h	5	Fin	15h

TABLE 3.2: Le backlog à la fin du sprint 5

3.1.2 Git

Afin de permettre une meilleure gestion du projet (travail parallèle, gestion de bugs, etc.) nous avons décidé d'utiliser le gestionnaire de version Git ainsi que la forge GitLab mise à la disposition des étudiants par le SIF. La prise en main fut facile, les membres du groupe ayant pratiquement tous déjà utilisé cet outil. Nous avons également utilisé le système de suivi de bugs intégré à Gitlab (cf. Figure 3.1).




<input type="checkbox"/> ID dans l'export	CLOSED
#29 1	updated 11 days ago
<input type="checkbox"/> Permettre au monde d'initialiser les zones	CLOSED
#28 1	updated 11 days ago
<input type="checkbox"/> Les appels de fonction récursifs font déborder la pile	
#27 improvement	updated 22 days ago
<input type="checkbox"/> Fonctions standard: un paramètre d'un mauvais type fait planter l'application	CLOSED
#26 assigned to  Plazas Adrien 1	updated 24 days ago
<input type="checkbox"/> String: supprimer les guillemets	CLOSED
#25 1	updated 23 days ago
<input type="checkbox"/> Utiliser pointeurs intelligents	CLOSED
#24 1 improvement	updated about a month ago
<input type="checkbox"/> Application "headless"	CLOSED
#23 1 improvement	updated 11 days ago
<input type="checkbox"/> Sérialisation / désérialisation	CLOSED
#22 1 improvement	updated 23 days ago
<input type="checkbox"/> Point : fuite mémoire	CLOSED
#21 assigned to  Plazas Adrien 1 bug model	updated 23 days ago
<input type="checkbox"/> Monde : ajouter des fonctions prédéfinies	CLOSED
#20 1 improvement	updated about a month ago
<input type="checkbox"/> Monde : ajouter des couleurs prédéfinies	CLOSED
#19 assigned to  Plazas Adrien 1 improvement	updated about a month ago

FIGURE 3.1 – Système de suivi de bugs de Gitlab

Notre organisation des branches fut la suivante :

- la branche master devait contenir une version fonctionnelle compilable ;
- des branches de développement étaient créées pour chaque nouvelle fonctionnalité, et n'étaient fusionnées sur la branche master que lorsqu'elles étaient pleinement fonctionnelles ;
- des branches spécifiques à chaque version ont été dérivées de master lors de l'officialisation desdites versions.

Nous utilisons également Gitg, pour avoir une meilleure vue de l'état de notre dépôt (cf. Figure 3.2).

3.1.3 Make

Make est un utilitaire de construction de fichiers qui nous a été utile tout au long du projet pour construire les applications (stibbons et stibbons-cli), les tests unitaires, ou encore la documentation L^AT_EX du projet. Il est également utile à l'installation des applications.

Il fonctionne en laissant l'utilisateur définir des règles de construction de fichiers en définissant les commandes nécessaires à celle-ci ainsi que les fichiers dont elle dépend.

Make est alors appelé à construire une cible (un fichier ou non), construisant son arbre de dépendances, vérifiant l'existence de ces dernières, les construisant ou reconstruisant au besoin. Ainsi Make permet d'éviter les compilations ou recompilations inutiles, accélérant et automatisant la construction de logiciels ou de documents.

3.2 Tests unitaires

3.2.1 CppUnit

CppUnit (ref. [Navarro, 2003]) est un outil permettant d'organiser des tests unitaires. On définit une classe de tests avec les attributs leurs étant nécessaires et des méthodes les réalisant. Cette classe est ensuite enregistrée dans le registre des tests pour être exécutée (cf. D.3).

CppUnit possède des macros pour simplifier les tests comme :

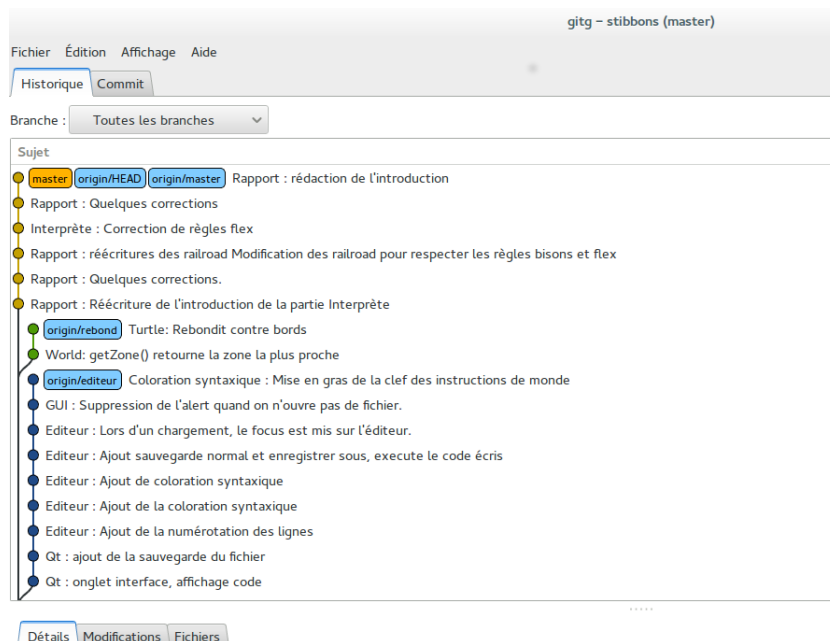


FIGURE 3.2 – Vue des branches dans Gitg

```
— CPPUNIT_ASSERT(test) ;
— CPPUNIT_ASSERT_EQUAL(v1,v2) ;
— etc.
```

Lors de l'exécution, il indique combien de tests ont réussi et échoué. Son écriture est simple, et son organisation permet une lecture rapide des tests.

3.3 C++11

C++11 est la version du standard C++ que nous avons choisi d'utiliser. En effet, cette version a vu apparaître de nombreux ajouts, comme par exemple les `std::thread` et les `std::mutex`, qui nous ont permis de gérer le multithreading sans avoir à nous soucier de leur implémentation (on n'a ainsi pas de problèmes de portabilité liés à ceci).

En outre, d'autres ajouts de C++11, tel que l'inférence de type (avec le mot-clef `auto`) ou les boucles traversant une collection (`for (auto i: collection)`), nous ont permis une écriture plus souple et plus moderne du code.

3.4 GDB

Le « GNU Project Debugger » (ref. [Stallman, 1988]) est un programme de débogage permettant notamment de tracer les appels de fonctions et de spécifier des conditions d'arrêts lors de l'exécution d'un programme. Il peut donc permettre d'avancer pas à pas dans un programme et de repérer l'endroit où un bogue se situe. Il est très utilisé pour résoudre les erreurs de segmentation grâce à sa précision de traçage : la ligne exacte de l'erreur est fournie.

3.5 Outils d'analyse

3.5.1 Flex

Flex est une version libre de l'analyseur lexical Lex (ref. [Paxson, 2014]). Il est généralement associé à l'analyseur syntaxique GNU Bison, la version GNU de Yacc. Il lit les fichiers d'entrée

donnés pour obtenir la description de l'analyseur à générer. La description est une liste de paires d'expressions rationnelles et de code C, appelées règles.

Un fichier Flex est composé de plusieurs parties. La première contient une partie optionnelle de définition, encadrée par les symboles `%{ %}` (cf. Listing 3.1), ainsi que des options pour Flex (cf. Listing 3.2). La seconde partie est une partie obligatoire de règles, commençant par `%%` (cf. Listing 3.3), tandis que la dernière partie est une nouvelle partie optionnelle, débutée par `%%`, pouvant contenir des fonctions C/C++ définies par l'utilisateur (cf. Listing 3.4).

```
%{
    int yyFlexLexer::yywrap() {
        return 1;
    }
}%
```

Listing 3.1 – Partie définition d'un fichier Flex

```
%option c++
%option nodefault
```

Listing 3.2 – Options de Flex

```
%%
#([a-f0-9]{6}|[a-f0-9]{3}) {
    pylval->v=make_shared<stibbons::Color>(yytext);
    return yy::parser::token::COLOR;
}
```

Listing 3.3 – Partie règles de Flex

```
%%
int main() {
    // ...
}
```

Listing 3.4 – Partie fonctions de Flex

La transformation en code C++ se fait par compilation via l'appel à l'application `flex -+ exemple.1+`. La fonction d'analyse ainsi générée se nomme `yylex()`. Il faut par la suite penser à compiler le programme en liant la bibliothèque Flex via le flag `-lfl`.

3.5.2 Bison

GNU Bison est une version de Yacc (ref. [bis, 2015]), un outil d'analyse syntaxique (cf. 5.2). Il génère un analyseur syntaxique ascendant utilisant un automate à pile (dérivation à droite, remplaçant le symbole non terminal le plus à droite).

Son fonctionnement est le suivant : à chaque règle de grammaire, on associe des actions (instructions d'un langage). L'analyseur généré essaie de reconnaître un mot du langage défini par la grammaire et exécute les actions pour chaque règle reconnue.

Comme pour Flex, un fichier Bison est composé de trois parties : la première partie, facultative, contient une liste de définition C/C++, d'options Bison ainsi que de définition de jetons (cf. Listing 3.5), la seconde partie, contenue entre `%%`, contient les règles (cf. Listing 3.6) et une dernière optionnelle de C/C++.

```
%skeleton "lalr1.cc"
%defines
%locations
```

```

%parse-param { stibbons::FlexScanner &scanner }
%parse-param { stibbons::TreePtr t }
%parse-param { stibbons::TablePtr w }
%lex-param { stibbons::FlexScanner &scanner }

%code requires {
    namespace stibbons {
        class FlexScanner;
    }

    std::string toString(const int& tok);
}

%token IF "if"
%token ELSE "else"
%token FCT "function"

```

Listing 3.5 – Definition C++ en bison

```

%%

//Storage of conditionnal expression
selection : IF expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
}
| IF expr statement ELSE statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->addChild($5);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
};

%%

```

Listing 3.6 – Règles de grammaire en bison

Les différents types de jeton sont déclarés via l'instruction `%token NOM_DU_JETON` dans la première partie. On peut également définir le type C/C++ de la valeur du jeton via l'instruction `%union`, ou en redéfinissant la macro `YYSTYPE` (cf. 3.7 et 3.8).

```

%union{
    stibbons::Value* v;
    stibbons::Tree* tr;
}

```

```
| }
```

Listing 3.7 – Exemple du type des valeurs des jetons avant la version 0.3

```
| #define YYSTYPE struct { stibbons::ValuePtr v; stibbons::TreePtr tr;  
|     int tok; }
```

Listing 3.8 – Exemple du type des valeurs des jetons en version 1.0

Si on veut un analyseur syntaxique en C++, il faut utiliser un squelette de parseur C++ en utilisant soit l'option bison `-skeleton=lalr1.cc`, soit en utilisant la directive `%skeleton "lalr1.cc"`.

La transformation en code C++ se fait par compilation via l'appel à l'application `bison -ydt exemple.y+`. La fonction d'analyse ainsi générée se nomme `yyparse()` et fait appel en interne à `yylex()`.

3.6 Qt

Qt est un framework d'application multi-plateformes écrit en C++ principalement utilisé pour la création d'interfaces graphiques.

3.6.1 Multi plateforme et multi langages

Qt est utilisable sur de nombreuses plateformes telles que Windows, Mac OS X, X11, Wayland, Android ou iOS.

De plus, bien que Qt soit développé en C++, ce n'est pas le seul langage depuis lequel il est utilisable, on trouve notamment des liaisons pour Python, JavaScript, Go, Ruby, Haskell ou encore Ada.

3.6.2 Modules

Qt comprend de nombreux modules afin d'aider tant que possible le développement d'applications. On peut particulièrement citer :

Core : une implémentation des types de base (`QString`, etc.), conteneurs, parallélisme, entrées-sorties, système d'événements, etc. ;

Widgets : des widgets pour le développement d'interfaces graphiques ;

Network : support de divers protocoles réseau (TCP, UDP, HTTP, SSL, etc.) ;

Multimedia : lecture audio et vidéo ;

SQL : accès à des bases de données comprenant SQL ;

WebKit : moteur de rendu HTML ;

3.6.3 Concepts fondamentaux

Widgets et layouts

Qt propose un système de widgets complet et puissant. Il propose de nombreux widgets classiques tels que des boutons, des choix à puce, des onglets, des étiquettes, des images, etc.

Pour Qt, tout widget peut contenir des enfants et les arranger selon une disposition qui lui est affectée. Un widget n'ayant pas de widget parent sera considéré comme étant une fenêtre. Son fonctionnement est ainsi assez différent de son concurrent Gtk+.

Signaux et slots

Qt propose également un système de signaux et de slots permettant d'implémenter le modèle observateur de manière efficace.

Ainsi un widget peut émettre des signaux contenant ou non des données (par exemple, pour signaler le changement de valeur d'une entrée) et un autre widget peut réceptionner ce signal dans un de ses slots, l'exécutant alors.

Une extension à C++

Qt propose une extension à C++ : il y ajoute des mots-clés pour permettre de simplifier la définition d'objets descendants de `QObject`, tout particulièrement en spécifiant un ensemble de slots d'une certaine visibilité. Ainsi lors de la déclaration d'une classe, il est possible de déclarer une liste de slots publics ou de signaux en les précédant des mentions `public slots` et `signals`, respectivement. Le Meta-Object Compiler de Qt est alors utilisé pour convertir ces définitions en C++ classique à la compilation.

Qt propose également un système permettant d'embarquer des ressources (images, sons, etc.) directement dans le binaire produit via la définition d'un fichier de collection de ressources (`.qrc`) et l'utilisation d'un Ressource Compiler.

qmake est un générateur de Makefiles permettant de simplifier l'utilisation du Meta-Object Compiler et du Ressource Compiler.

3.6.4 Outils

Qt permet de développer des interfaces dans un langage déclaratif basé sur XML. Pour utiliser ce puissant atout, il est conseillé d'utiliser Qt Designer, une application permettant de dessiner des interfaces graphiques de manière intuitive, en plaçant manuellement les widgets les uns dans les autres.

Qt Creator va encore plus loin puisqu'il est un IDE assez complet centré sur le développement d'applications avec Qt. En effet, il permet de gérer des projets, d'éditer du code, de concevoir des interfaces comme Qt Designer et même de créer des slots de manière graphique. Bien entendu il permet également de compiler, exécuter et déboguer le projet.

3.6.5 Dessiner avec Qt

Il est possible de réaliser des widgets personnalisés en redéfinissant la méthode `paintEvent` d'un widget et en utilisant la classe `QPainter` pour dessiner sur le widget.

Une instance `QPainter` est liée à un widget et propose diverses méthodes permettant de dessiner :

- des lignes ;
- des arcs de cercle ;
- des polygones ;
- des ellipses ;
- des images ;
- du texte.

3.7 Latex

3.7.1 Généralités

L^AT_EX est un langage de rédaction de document qui force à avoir une structure sur la forme et le contenu. Il est notamment utilisé lors d'écritures de documents scientifiques car son écriture de contenus complexes (équations, bibliographie, etc.) se manie facilement. Contrairement à

d'autres logiciels de rédaction tel que LibreOffice, OpenOffice, etc., \LaTeX n'est pas de type WYSIWYG (What You See Is What You Get). Il faut donc expliciter la mise en page du document, d'où sa catégorie de langage.

\LaTeX possède des implémentations libres, dont notamment TeX Live.

3.7.2 Beamer

Beamer est un paquet de \LaTeX spécialisé pour la création de présentations sous forme de diapositives. Plusieurs thèmes existent pour la mise en forme et, comme \LaTeX , Beamer n'est pas de type WYSIWYG.

3.8 JSON Spirit

JSON Spirit (ref. [Wilkinson, 2014]) est une bibliothèque C++ qui permet de manipuler des fichiers JSON avec du C++.

Il utilise des structures de données C++ comme les `std::vector` ou les `std::map`, pour stocker les objets JSON.

C'est un outil nouveau, mais assez puissant. Il n'est pas difficile à prendre en main : une classe `Value` existe, et représente n'importe quels types de données (tableaux, objets, etc.) et sert de base à la construction d'objet comme les `Array` de JSON Spirit (un `vector` de `Value`), ou les `Object` (un `vector` de `std::pair` C++).

Grâce à ces emboîtements, on peut représenter un fichier JSON avec exactitude.

Chapitre 4

Modèle

4.1 Les classes d'agents

Sprints 1 et 2

Lors du premier sprint, nous avons commencé par mettre en place la base du programme. Nous avons prévu de réaliser l'UML présent en figure 4.1.

Pour le modèle, il s'agissait de créer les classes :

- **Turtle**, qui représente une tortue;
- **Point**, pour stocker les coordonnées d'une tortue;
- **Line**, qui est composé de points, et qui permet aux tortues de laisser une trace (instruction `pen_down`);
- **World**.

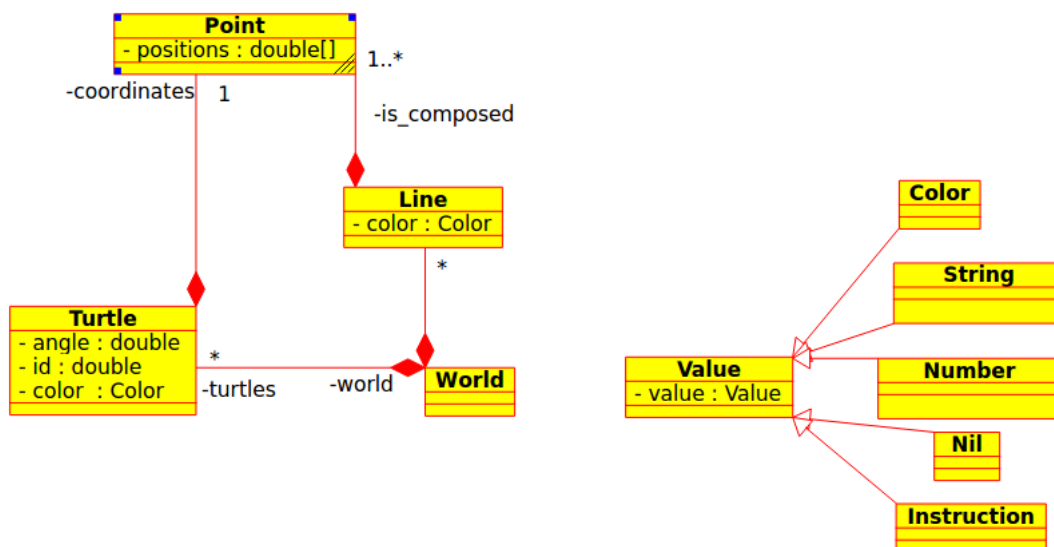


FIGURE 4.1 – UML prévisionnel de la version 0.1

La classe **World** représente le monde, contenant la liste des tortues, et les lignes tracées par celles-ci, et communique avec l'interface pour l'affichage. À la fin du sprint 1, on pouvait voir une tortue avancer et tracer une ligne sur son passage. Nous avons réalisé le schéma 4.2. Il respecte la version prévue, en dehors de la classe **Instruction**, qui n'était finalement pas nécessaire, les instructions étant des méthodes de **Turtle**.

Lors du second sprint, nous avons ajouté une classe **Agent**, super-classe de **World**, **Turtle** et **Zone**; en effet, elles sont toutes les trois des agents, et ont donc des comportements similaires

(cf. Figure 4.3).

Ces classes contiennent chacune un parent, une liste d'enfants, et des propriétés. Les propriétés sont des variables définies par l'utilisateur lors de la définition du code de l'agent, donc surtout utile pour les tortues.

De plus, le monde a une taille et deux listes d'espèce de tortues (**Breed**) : les espèces nommées et les anonymes.

Comme le montre le listing 4.1, on peut créer des tortues nommées ou pas. Une tortue anonyme a son corps défini à la création de l'agent (lors du **new agent**) tandis que les tortues nommées on leur corps défini lors de la création du type d'agent (instruction **agent <nom>**).

```
agent listener () {  
    fd 2  
}  
  
new listener ()  
  
new agent {  
    lt 30  
    fd 5  
}
```

Listing 4.1 – Nommage lors de la création d'une tortue

Sprints 3 et 4

Lors du troisième sprint, nous avons mis en place des pointeurs intelligents dans toutes nos classes (cf. Figure 4.4). Le but de ce sprint était la mise en place de la communication entre les agents : les tortues devaient pouvoir communiquer via les zones par modification de leurs propriétés, et elles devaient aussi pouvoir communiquer entre elles grâce à des instructions comme **send** et **recv**.

L'étape suivante était d'ajouter des fonctionnalités telles que la maîtrise du temps et l'exportation du modèle. Ces ajouts ne provoquent pas de changements majeurs du côté du modèle, si ce n'est quelques méthodes dans les classes **World**, **Turtle** et **Zone** pour l'export du modèle. Celui-ci consiste à créer une sauvegarde de l'état du modèle à un instant **t** dans un fichier JSON grâce à la bibliothèque JSON Spirit (cf. Listing D.4). Cela permettra ensuite, par exemple en passant par une transformation en CSV, d'avoir des tableaux avec toutes les données, ce qui offre la possibilité d'avoir des diagrammes de l'évolution du monde.

La maîtrise du temps se fait grâce à un bouton pause, qui arrête les threads s'exécutant, ou par un slider qui permet de ralentir ou de diminuer la vitesse.

4.2 Les types

Sprints 1 et 2

Lors du premier sprint, nous avons mis en place un certains nombre de types (cf. Figure 4.2), comme **Color**, représentant les couleurs, ou encore **Nil**, représentant la valeur nulle. Ils héritent de **Value**, une classe abstraite qui contient une valeur et ses accesseurs.

Lors du second sprint, les types **Stibbons** sont les mêmes mais leurs définitions se sont un peu complexifiées, en passant par une classe **SimpleValue** pour la mise en place des mutex. Une énumération des types **Stibbons** existe, elle est utilisée avec la méthode **getType()** pour pouvoir connaître le type d'une valeur. Pour que l'utilisateur puisse écrire des fonctions dans le code, nous avons ajouté une classe **Function**, qui stocke un arbre abstrait, contenant le code de la

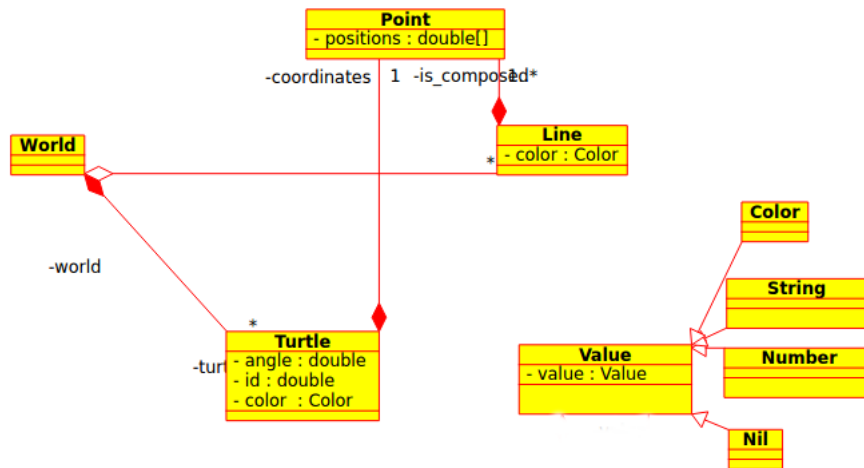


FIGURE 4.2 – UML de la version 0.1 réalisée

fonction déjà analysé. Des mutex ont également été ajoutés dans toutes les classes pour assurer que les objets soient thread-safe (cf. 4.3).

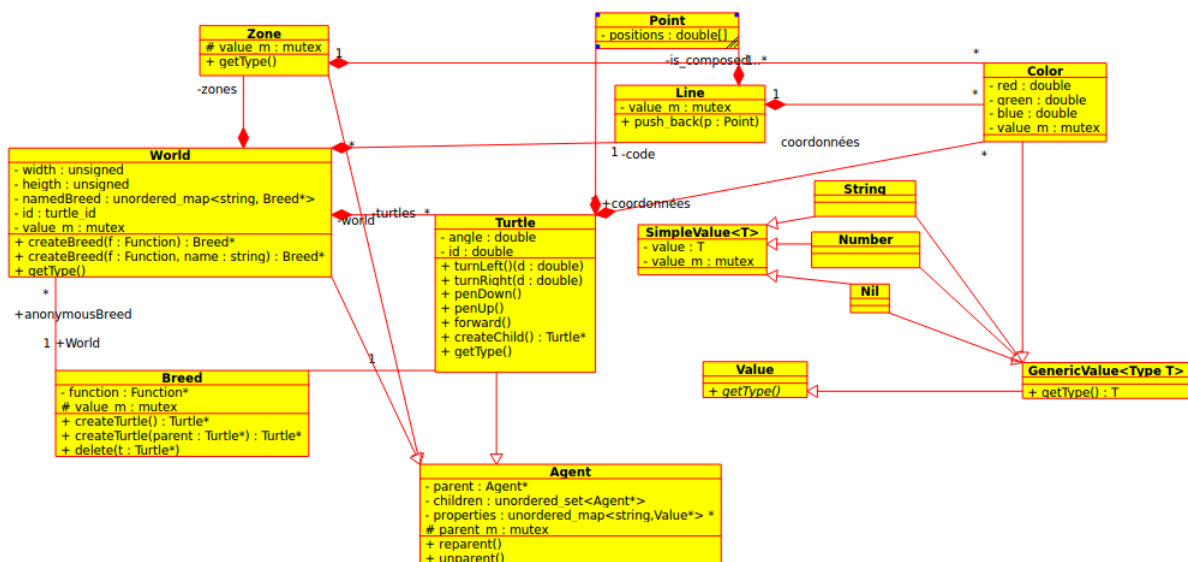


FIGURE 4.3 – UML version 0.2

Sprints 3 et 4

Lors du troisième sprint, nous avons ajouté une sous-classe de `Function`, `userFunction`, qui représente les fonctions créées par l'utilisateur (cf. Figure 4.4). Nous avons également créé des fonctions standard comme `ask_zone`, qui permettent de donner des ordres aux zones.

L'ajout de `Table` représentant l'unique type de conteneurs, les tableaux, fait aussi parti de ce sprint. Nous les écrivons à la façon de PHP (cf. Listing 4.2).

```
a = 12

t = {18, red, "bla"}
v = { "bla" : "blou", "blue" : blue, a : 29 }
```



```

println(t[2])
t[2] = 32
println(t[2])
v[] = 48
println(v)

u = 5 new agent {
  while true fd 20
}

```

Listing 4.2 – Syntaxe des tables en Stibbons

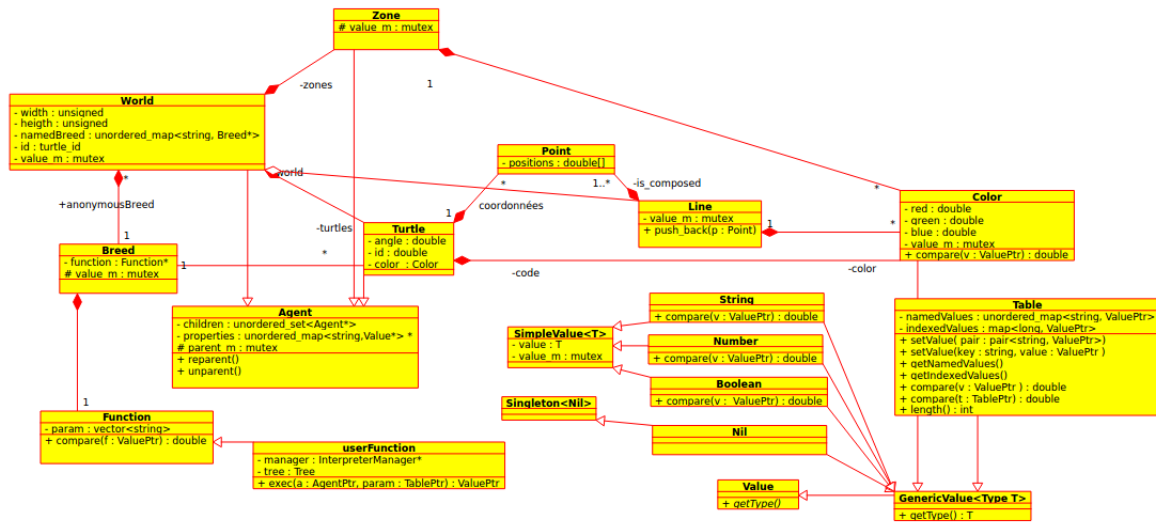


FIGURE 4.4 – UML version 0.3

Nous avons mis en place des mutex récursifs lors du quatrième sprint. Ils permettent à un thread de verrouiller plusieurs fois la même ressource qu'il a déjà verrouillée. C'est la seule différence avec le mutex normal.

Ces mutex permettent d'éviter des blocages dans certaines situations (fonctions récursives, etc.). L'UML 4.5 est l'état final du modèle, le sprint 5 n'y ayant apporté aucune modification.

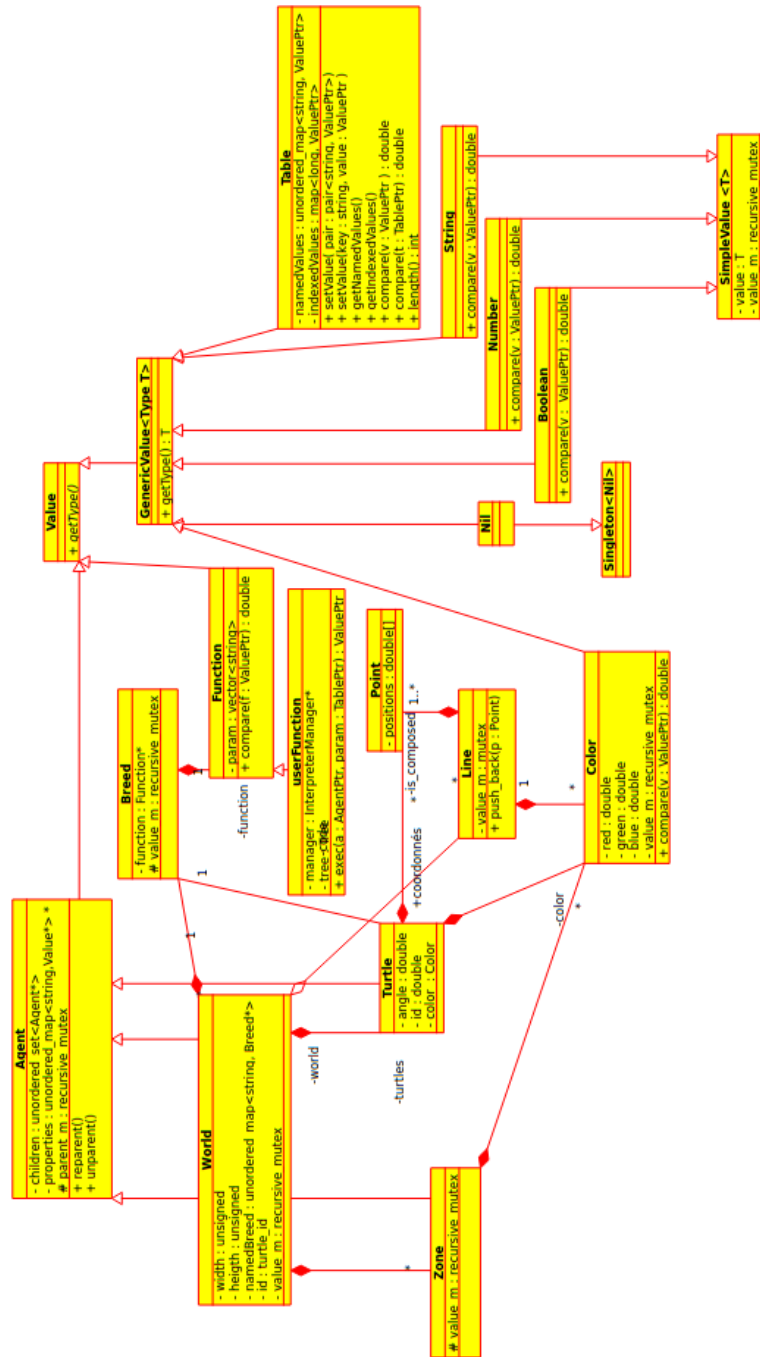


FIGURE 4.5 – UML version 0.4

Chapitre 5

Interprète

L'application Stibbons fournit un interprète (ou interpréteur) pour le langage du même nom. Cette interprétation du code se déroule en deux phases : une de compilation lors du chargement du code, et une d'interprétation de l'arbre abstrait (généré durant la première phase) lors de l'exécution.

La première phase peut elle-même être découpée en deux parties :

- l'analyseur lexical qui « lit le flot de caractères qui constituent le programme source et les regroupe en séquences de caractères significatives appelées *lexèmes*. » [Aho et al., 2007] ;
- l'analyseur syntaxique qui, à partir des lexèmes, génère un arbre abstrait qui pourra par la suite être analysé pour être interprété.

L'interprétation quant à elle se déroule lors de l'analyse sémantique de l'arbre abstrait, qui exécute les opérations contenues dans les nœuds.

5.1 Analyseur lexical

L'analyse lexicale vise à produire un flot de jetons qui pourront être analysés par l'analyseur syntaxique. Ces jetons sont des paires composées d'un type de jeton et de la valeur du lexème (par exemple, l'analyse du lexème 12 va générer le jeton `<NUMBER,12>` dans notre cas). Certains lexèmes peuvent générer des jetons qui n'ont pas de valeur (par exemple, le lexème `(` va entraîner la génération du jeton `<(>`).

Nous avons fait le choix d'utiliser l'outil Flex pour notre projet (cf. 3.5.1).

5.1.1 Jetons

Notre analyseur lexical a dans un premier temps généré un nombre limité de jetons. Ainsi, en version 0.1, nous générions seulement 26 jetons différents (dont 5 jetons de littéraux), contre 40 jetons en version 1.0 (dont 7 jetons de littéraux). Ces jetons sont définis dans le code source Bison (cf. Listing D.2) et leurs valeurs ont un type C++ défini par la structure listing 5.1. Dans cette structure, l'attribut `tok` correspond au type du jeton (défini par une énumération plus tard) tandis que l'attribut `v` correspond à une valeur. En effet, notre langage ayant un typage dynamique, toutes les valeurs ont un type statique de type `Value`. Ainsi, grâce au polymorphisme, le jeton aura une valeur qui pourra être de type dynamique `Number`, `String`, `Boolean`, etc. tout en gardant un type statique de `Value`. C'est lors de l'analyse sémantique (cf. 5.3) que le type réel du jeton est analysé.

```
struct {  
    stibbons::ValuePtr v;  
    stibbons::TreePtr tr;  
    int tok;
```

```
| };
```

Listing 5.1 – Type des valeurs des jetons

5.1.2 Fonctionnement

Par défaut, l’analyseur lexical généré par Flex utilise des variables globales et n’est pas réentrant. Dans l’optique d’obtenir un analyseur réentrant, nous avons utilisé l’option `%option c++` et établi le modèle 5.1.

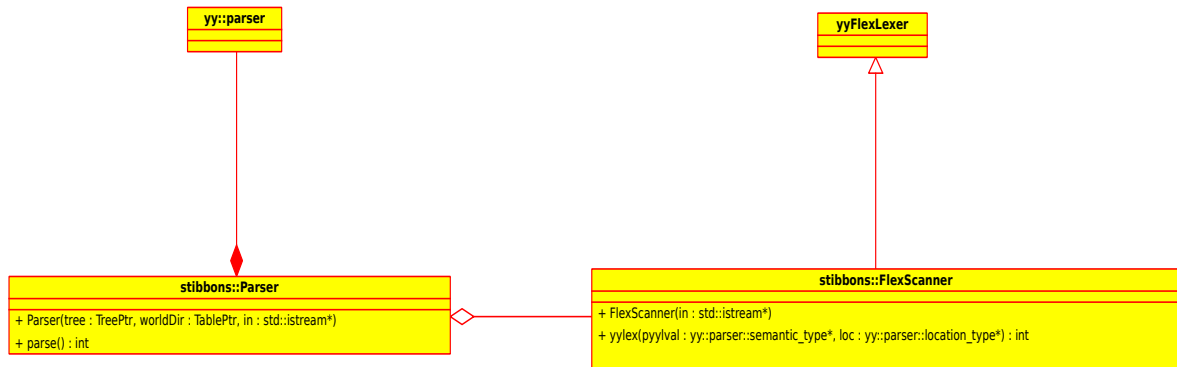


FIGURE 5.1 – UML des analyseurs lexical et syntaxique

Chaque appel à la méthode `yylex()` de `FlexScanner` génère un nouveau jeton à partir du flux d’entrée, passé en paramètre lors de la construction de cet objet. Cette méthode analyse le flux à partir des règles définies dans le fichier `lexer.l+` (cf. Listing D.1); ainsi, les instructions définies pour chaque règle sont effectuées quand une chaîne de caractères correspondante à la règle est détectée. Dans l’extrait 5.2, lorsque l’analyseur détecte le caractère `#` suivi d’une séquence de 3 ou 6 nombres hexadécimaux, un appel au constructeur de `Color(string)` est effectué. Ce dernier crée une couleur à partir d’une chaîne de caractère respectant les codes de couleur html.

```

#([a-f0-9]{6}|[a-f0-9]{3}) {
    pyylval->v=make_shared<stibbons::Color>(
        yytext);
    return yy::parser::token::COLOR;
}
  
```

Listing 5.2 – Exemple de séquence d’instructions lors de la détection d’une couleur

De plus, un appel à la méthode `step()` du paramètre `loc` (cf. Figure 5.1) est effectué au début de chaque appel à `yylex()`, et permet de faire avancer la position actuelle de la longueur du lexème détecté, et un appel à sa méthode `lines()` est effectué lors de la détection d’un retour à la ligne, afin de faire avancer de `n` lignes la position actuelle (avec `n` le nombre de retour à la ligne détecté). Les différentes règles Flex peuvent être consultées au listing D.1, ou de façon plus lisible dans l’annexe A.1.1.

5.2 Analyseur syntaxique

L’analyse syntaxique permet de vérifier que la structure d’un programme est bien en accord avec les règles de grammaire du langage. Par exemple, la grammaire de notre langage comporte une règle, qui indique qu’un appel de fonction sans paramètre, est constituée du jeton `<ID>` suivi

des jetons $\langle \rangle$ et $\langle \rangle$. Le but de notre analyse ici est double : vérifier que le programme est bien un programme de notre langage valide, mais également générer un arbre abstrait qui pourra facilement être analysé par notre analyseur sémantique.

Nous avons fait le choix d'utiliser GNU Bison (cf. 3.5.2) dans notre projet.

5.2.1 Arbre abstrait

Un arbre abstrait est une structure d'arbre dont chaque nœud feuille représente les opérandes des opérations contenues sur les autres nœuds. Cet arbre peut être considéré comme une forme de code intermédiaire, et peut être interprété très facilement en parcourant, pour chaque nœud, les sous-arbres et en appliquant l'opération du nœud courant.

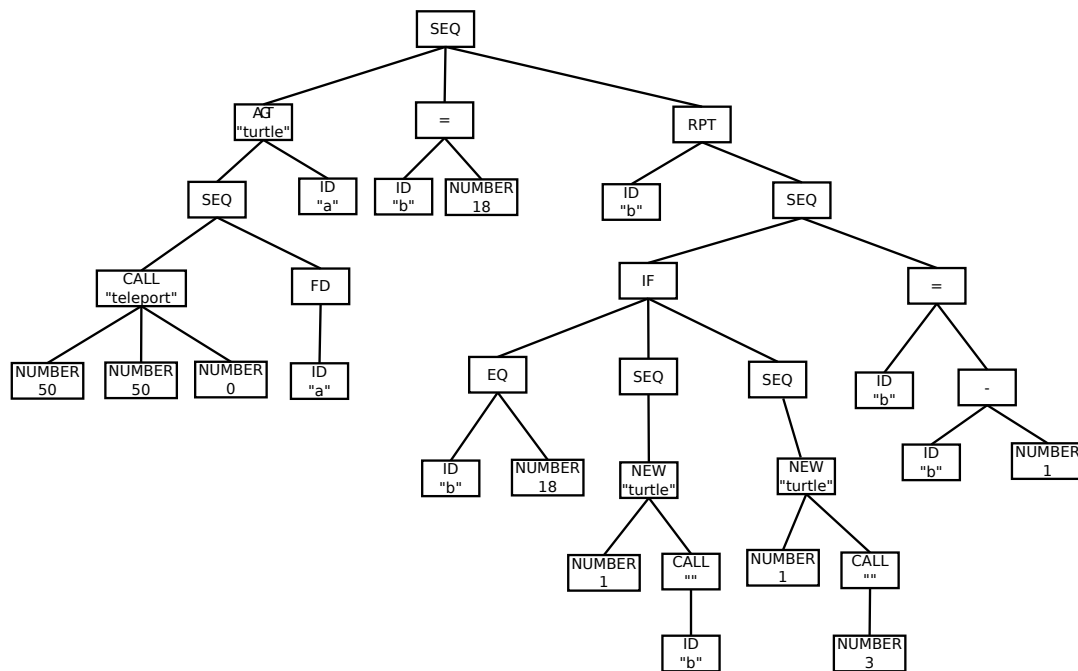


FIGURE 5.2 – Arbre abstrait généré lors de l'analyse du code 5.3

```

agent turtle (a) {
  teleport(50,50,0)
  fd a
}

b = 18
repeat b {
  if(b == 18) {
    new turtle(b)
  }
  else {
    new turtle(3)
  }
  b = b - 1
}

```

Listing 5.3 – Exemple de code Stibbons

L'avantage de générer un tel arbre est qu'il est bien plus rapide d'effectuer un parcours d'arbre à chaque interprétation plutôt que de refaire une analyse syntaxique à chaque fois.

5.2.2 Fonctionnement

L'écriture du fichier Bison est assez semblable au fichier Flex. En effet, on peut associer à chaque règle une liste d'instructions qui vont être exécutées lors de la détection de la règle. Ces instructions peuvent en outre être des affectations de valeur sémantique à une expression. Ainsi, dans l'extrait de code 5.4, la valeur sémantique d'une sélection est un arbre dont les enfants sont dans l'ordre : l'expression de test, la séquence d'instructions à exécuter si la condition est vraie et, optionnellement, la séquence d'instructions à exécuter si la condition est fausse.

```
selection : IF expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
}
| IF expr statement ELSE statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->addChild($5);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
};
```

Listing 5.4 – Cas du IF en Bison

Bison génère à partir de ces règles un analyseur syntaxique LALR (*look-ahead left-to-right rightmost derivation*). L'analyse du code et la génération de l'arbre sont lancées par un appel à la méthode `stibbons::Parser::parse()` qui va elle-même faire appel à la méthode `yy::parser::parse()`. L'ensemble du code va être analysé, cette méthode se chargeant de faire appel à la méthode `stibbons::FlexScanner::yylex()`, générant et analysant les jetons en une seule passe.

5.3 Analyseur sémantique

L'objectif de l'analyseur sémantique est l'interprétation pure et simple de l'arbre abstrait précédemment généré. En effet, le but ici est l'exécution du code écrit en Stibbons, et l'interprète prend ainsi en paramètre un arbre ainsi qu'un agent sur lequel exécuter cet arbre. Il effectue un parcours en profondeur de l'arbre, se rappelant récursivement sur chaque nœud fils de l'arbre.

5.3.1 Fonctionnement

Lors de l'analyse d'un jeton, qui correspond à un nœud de l'arbre syntaxique, on analyse d'abord ses enfants (s'il y en a) puis ensuite le contenu du nœud courant. Par exemple, pour le test d'égalité `1 == 2`, le nœud `<EQ>` aura pour enfants les sous-arbres `<NUMBER,1>` et `<NUMBER,2>`. Ces deux derniers seront interprétés en premier et renverront leurs valeurs sémantiques (en

l'occurrence, des `stibbons::Number` ayant respectivement pour valeur 1 et 2), puis le test `==` sera appliqué sur ces résultats.

Nous avons dû faire un certain nombre de choix sémantiques lors de l'élaboration de notre langage, et en voici quelques uns particuliers :

affectation Lors d'une affectation, on modifie en priorité les paramètres de l'agent ou de la fonction courante, si l'identifiant existe dans la table des paramètres, sinon, on modifie la propriété correspondante de l'agent.

identifiant Lorsque l'on accède à la valeur d'un identifiant, on recherche tout d'abord cet identifiant dans les paramètres de l'agent courant. S'il n'existe pas, on le recherche dans les propriétés de l'agent courant. Ainsi, la portée des « variables » est exclusivement locale à l'agent. On peut explicitement accéder à une propriété d'un autre agent, via l'opérateur « `.` ».

fonction Les fonctions sont des valeurs. Ainsi, tout comme les valeurs, leurs créations ont un comportement similaire à une affectation et leurs accès également (espace de noms des fonctions et des variables communs).

agent La création d'un nouveau type d'agent, où qu'elle soit faite, est liée au monde et est, par conséquent, accessible depuis tous les autres agents.

5.3.2 Fonctionnalités

Sprints 1 et 2

Les deux premiers sprints ont été assez conséquents au niveau du nombre de fonctionnalités ajoutées. En effet lors du sprint 1, on pouvait déjà effectuer les opérations élémentaires sur une tortue, telles que avancer, tourner, écrire, etc. De plus, les opérations arithmétiques ainsi que le support des nombres ont été faits. En effet, il était nécessaire de gérer les nombres de manière à pouvoir indiquer à la tortue la distance de laquelle elle devait avancer.

Lors du deuxième sprint sont apparus les conditionnelles, ainsi que les boucles, les comparaisons d'ordre, les booléens et de nouveaux types (couleurs, chaînes de caractères, etc.). Sont également apparues la création de nouveaux agents dans le code et les fonctions sans paramètres. Ce sprint fut déjà une version bien avancée de notre programme.

Sprints 3 à 5

Les sprints suivants furent plus légers. Non pas parce qu'il y avait moins de travail, car la gestion de l'interpréteur fut remaniée à ce moment là, mais car il y avait moins de fonctionnalités à ajouter. En effet, à partir du sprint 3, les fonctionnalités suivantes ont été rajoutées :

- l'accès à la parenté et aux zones ;
- l'ajout des tables et de boucles dédiées (`for`) ;
- la gestion de la vitesse et de la pause ;
- l'ajout des communications avec le `send` et le `recv`.

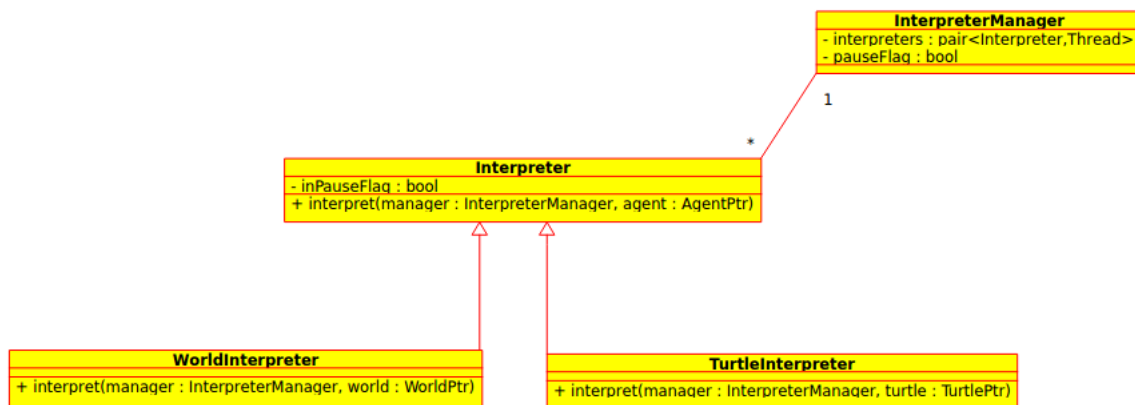
5.3.3 Organisation

L'analyseur sémantique a d'abord été implémenté par l'unique classe `Interpreter`. Cette dernière implémentait tous types d'actions à effectuer pour n'importe quel type d'agent. Cependant, lors de l'arrivée de la fonctionnalité d'ajout d'un nouvel agent (`new agent`), nous nous sommes aperçus qu'il était plus intéressant d'avoir un interpréteur par type d'agent, ou plus précisément un interpréteur pour le monde, un pour les tortues et un pour les actions communes aux deux types (cf. UML 5.3) ; nous avons alors créé les classes `WorldInterpreter` et `TurtleInterpreter`.

Ainsi, par exemple, si on demande à une tortue d'avancer en Stibbons (`fd 10`), alors c'est le `TurtleInterpreter` qui gèrera cette action. Par contre, si on effectue une affectation (`a = 10`) alors l'`Interpreter` affectera la valeur 10 à la propriété `a` de l'agent courant.

De manière parallèle, la création du monde s'effectuait dans l'`Interpreter`, puis dans le `WorldInterpreter`, il fut alors nécessaire de créer une classe qui gèrerait à la fois la création du monde et tout ce qui concernait l'application (la pause, le temps, les interpréteurs eux-mêmes). Nous avons alors décidé de créer la classe `InterpreterManager`. Cette classe connaît tous les interpréteurs, et permet de les prévenir d'une éventuelle pause du programme, de stocker les threads correspondants à chaque interpréteur, de créer un monde avec les directives choisies ; c'est une sorte de gestionnaire d'interpréteurs.

FIGURE 5.3 – UML de l'analyseur sémantique



Chapitre 6

Applications

6.1 Application graphique

6.1.1 Version 0.1

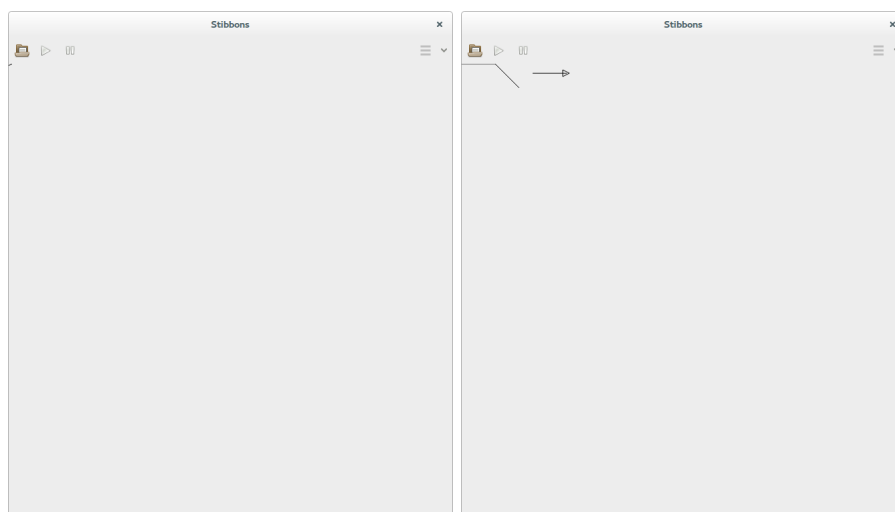


FIGURE 6.1 – Capture d’écran de la version 0.1

L’interface de la version 0.1 (cf. Figure 6.1) de Stibbons est composée d’une fenêtre séparée en une barre d’outil et une vue du monde. La barre d’outil contient un bouton permettant d’ouvrir un fichier contenant un programme Stibbons (Ctrl+O), un bouton permettant d’exécuter le programme ouvert, un bouton de pause non fonctionnel, un bouton ouvrant un menu proposant la boîte de dialogues « À propos » et de quitter l’application (Ctrl+Q).

Si la boîte de dialogue permettant d’ouvrir un programme est fermée sans avoir choisi de fichier, un message d’erreur l’indiquant est alors affiché.

Une tortue existe par défaut dans le monde, elle est située dans le coin supérieur gauche de la vue et est orientée vers la droite. La tortue est alors dessinée comme un triangle noir creux et elle peut dessiner des lignes noires.

L’analyseur syntaxique n’étant à ce moment pas réentrant, ouvrir plusieurs programmes les uns après les autres implique alors de redémarrer l’application.

6.1.2 Version 0.2

La version 0.2 (cf. Figure 6.2) apporte peu de modifications à l’application Stibbons elle-même, le changement le plus visible étant le placement temporaire de l’origine au centre de la vue, afin d’avoir une meilleure vue du programme s’exécutant.

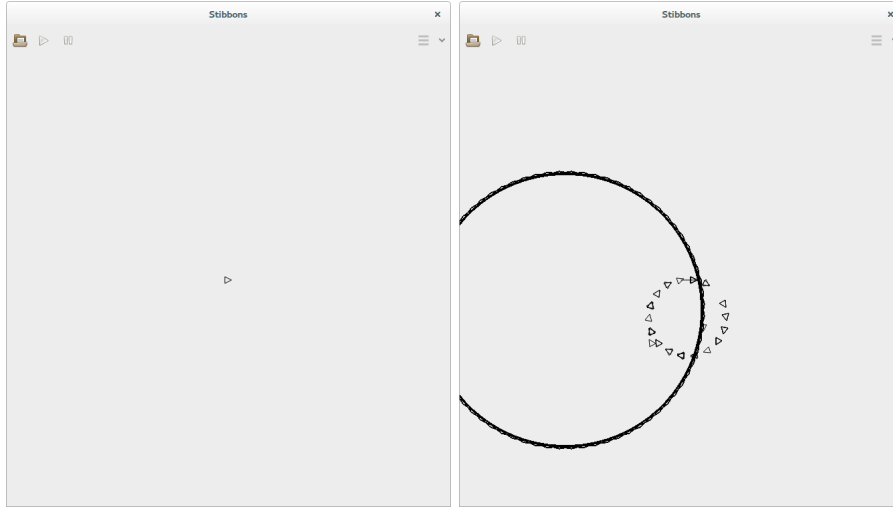


FIGURE 6.2 – Capture d’écran de la version 0.2

6.1.3 Version 0.3

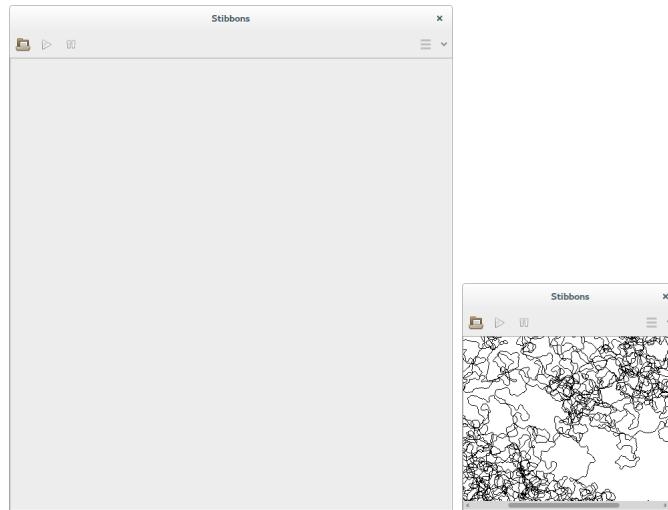


FIGURE 6.3 – Capture d’écran de la version 0.3

À partir de la version 0.3 (cf. Figure 6.3), les couleurs des tortues peuvent être modifiées et, afin de les rendre plus visibles, elles sont désormais dessinées comme des triangles pleins. La ligne dessinée par une tortue prend la couleur de cette dernière au moment de l’abaissement du stylo.

Le monde est désormais borné, c’est à dire que les tortues ne peuvent plus sortir du monde visible pour l’utilisateur, et est centré dans la vue, les zones le constituant sont désormais affichées, et leurs couleurs peuvent être modifiées. Si la vue est plus petite que le monde, des ascenseurs seront affichés.

Redessiner le monde prenait de plus en plus de temps au fur et à mesure que le nombre de lignes le constituant augmentait. Afin d’améliorer drastiquement ces performances de rendu, les lignes seront désormais dessinées sur un tampon et seules les nouvelles lignes auront donc besoin d’être dessinées sur ce dernier, avant qu’il soit reporté sur la vue.

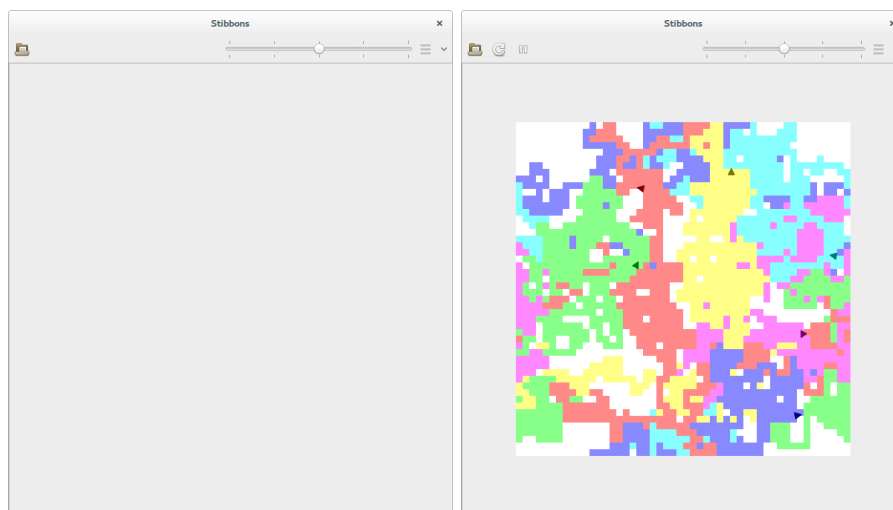


FIGURE 6.4 – Capture d’écran de la version 0.4

6.1.4 Version 0.4

À partir de la version 0.4 (cf. Figure 6.4), il est possible de modifier la vitesse d’exécution du programme, de le mettre en pause, de reprendre son exécution et de le redémarrer depuis le début (F5). De plus, l’analyseur syntaxique étant désormais réentrant, il est possible d’ouvrir plusieurs programmes les uns à la suite des autres.

Les boutons de démarrage et de mise en pause sont désormais confondus, et sont cachés, de même que le bouton de redémarrage, tant qu’aucun programme n’a été ouvert.

Une entrée a été ajoutée au menu, permettant d’exporter le modèle en cours d’exécution.

Si une tortue change de couleur alors qu’elle trace une ligne, le stylo avec lequel la tortue trace ses lignes changera de couleur en même temps.

Il est désormais possible de faire reboucler le monde sur lui même, cette option étant paramétrable pour chaque axe. Cela permet aux tortues, lorsqu’elles atteignent un bord, d’apparaître du côté opposé, comme si le monde était circulaire.

Le monde ne comporte plus de tortue par défaut et c’est désormais celui-ci qui exécute le corps principal du programme.

6.1.5 Version 1.0

La version 1.0 (cf. Figure 6.5) marque l’apparition d’un éditeur de programme Stibbons dans l’application elle-même. L’éditeur propose également une coloration syntaxique. L’ajout de cet éditeur implique que les boutons de démarrage, de pause et de redémarrage ne soient plus cachés par défaut car il est désormais possible d’exécuter un programme écrit sans en avoir chargé un avant.

De nombreux raccourcis clavier sont ajoutés, afin d’enregistrer le programme dans son fichier (Ctrl+S) ou dans un nouveau fichier (Ctrl+Shift+S), d’exporter le modèle (Ctrl+E) et de démarrer ou mettre en pause l’exécution du programme (Ctrl+Espace).

Si la version 0.4 a ajouté une option permettant au monde de reboucler selon chaque axe, la version 1.0 permet d’avoir des bords solides, faisant rebondir toute tortue les croisant.

6.2 Application en ligne de commande

Une application en ligne de commande a été ajoutée à la version 0.4. Elle permet d’exécuter des programmes Stibbons sans serveur graphique et à pleine vitesse.

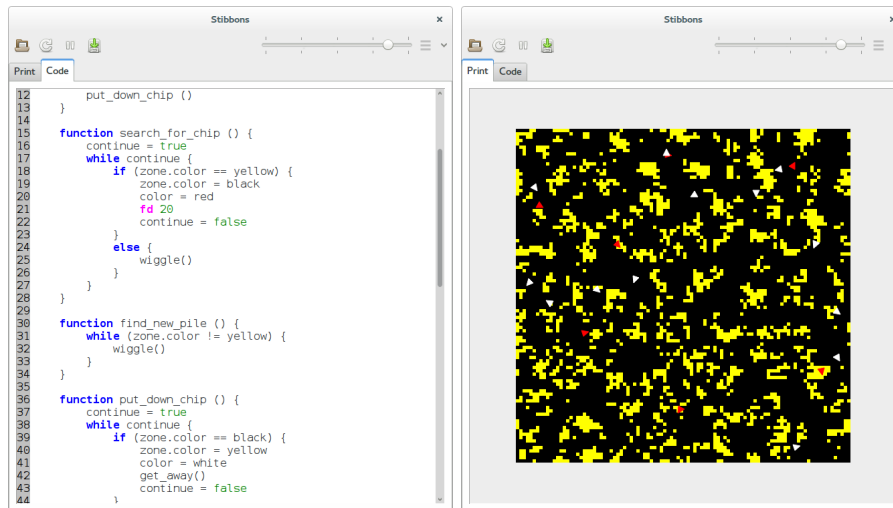


FIGURE 6.5 – Capture d’écran de la version 1.0

6.2.1 Utilisation

Utilisation : `stibbons-cli [options] fichier`

Options :

- h, --help** Affiche l’aide.
- v, --version** Affiche l’information de version.
- e, --export <secondes>** Exporte le modèle toutes les <secondes> secondes.
- p, --prefix <prefixe>** Préfixe les fichiers exportés avec <prefixe>.
- png** Génère une image PNG pour chaque export.
- no-json** N’exporte pas le modèle dans un fichier JSON.

Arguments :

fichier Le fichier de programme Stibbons à exécuter.

6.2.2 Fonctionnement

Le programme analyse la ligne de commande à l’aide de la classe `QCommandLineParser`, puis crée et exécute un objet représentant l’application selon les paramètres passés par l’utilisateur.

Lors de l’exécution de l’application, il est possible d’exporter le modèle à intervalle régulier grâce à l’option **--export**. Par défaut, le modèle est exporté dans un fichier JSON, mais il est également possible d’exporter un rendu du monde en PNG, ou encore de ne rien exporter.

Afin de pouvoir dessiner le monde dans un fichier image, la classe `WorldView`, widget de l’application graphique jusque là responsable de dessiner le monde sur lui même, a vu ses fonctionnalités de dessin d’un monde migrer vers la nouvelle classe `WorldPainter`, capable de dessiner un monde sur une surface. Cette nouvelle classe est désormais utilisée dans l’application graphique par `WorldView` pour se dessiner, et par l’application en ligne de commande pour dessiner sur une `QImage`.

Chapitre 7

Conclusion

Durant quatre mois, nous avons réalisé ce projet que nous avions imaginé. Ce projet a été positif à bien des égards. En effet, il nous a tout d'abord permis de pratiquer une méthode agile, ce qui était une bonne expérience et, avec le recul, un bon choix, car nous avons un projet fonctionnel toutes les deux semaines. Nous avons pu en outre grâce à cette méthode ajouter facilement de nouvelles fonctionnalités en cours de route, avec notamment l'application en ligne de commande, complémentaire à notre programme principal.

Ce fut également une expérience profitable car nous avons pu voir l'importance de la communication sur ce type de développement. En effet, les cycles étant très courts, il était important de communiquer rapidement et clairement pour éviter de prendre du retard. Si nous avons parfois dépassé légèrement le délai (quelques heures de retard sur le premier sprint), nous avons tout de même su gérer notre temps, ce qui nous a permis d'être dans les temps sur l'ensemble du projet.

Nous avons également pu grâce à ce projet travailler avec des outils nouveaux (Qt, JsonSpirit, `std::thread`) et nous améliorer sur d'autres outils (Flex, Bison, Pointeurs intelligents).

Bien que la 1.0 soit une version fonctionnelle, il y a cependant des choses à améliorer. La première, et sans doute la plus importante, est le fait que notre parcours d'arbre se fait de façon récursive. Par conséquent, les appels récurifs en Stibbons vont impacter directement la pile d'exécution de l'interprète, et nous limite beaucoup. De plus, les tortues évoluant chacune dans un thread séparé, on est relativement vite limité par le nombre de tortues pouvant évoluer dans notre programme (changement de contexte lourd). Il y a également bon nombre de fonctionnalités que nous pourrions ajouter, comme les entrées dans l'interface, un peu à la manière de NetLogo, qui permettraient de modifier des propriétés de façon interactive avec l'exécution, ou encore l'affichage de la sortie standard dans un cadre du programme, voire dans des petites bulles au dessus des tortues.

Nous retiendrons néanmoins de notre projet un bilan positif. En effet, nous avons créé un langage, le Stibbons, ainsi que son interprète et une application graphique permettant de suivre l'évolution du modèle.

Table des figures

3.1	Système de suivi de bugs de Gitlab	13
3.2	Vue des branches dans Gitg	14
4.1	UML prévisionnel de la version 0.1	20
4.2	UML de la version 0.1 réalisée	22
4.3	UML version 0.2	22
4.4	UML version 0.3	23
4.5	UML version 0.4	24
5.1	UML des analyseurs lexical et syntaxique	26
5.2	Arbre abstrait généré lors de l'analyse du code 5.3	27
5.3	UML de l'analyseur sémantique	30
6.1	Capture d'écran de la version 0.1	31
6.2	Capture d'écran de la version 0.2	32
6.3	Capture d'écran de la version 0.3	32
6.4	Capture d'écran de la version 0.4	33
6.5	Capture d'écran de la version 1.0	34

Liste des tableaux

3.1	Le backlog au début du sprint 2	11
3.2	Le backlog à la fin du sprint 5	12

Listings

2.1	Procédure en Logo	7
2.2	Boucle en Logo	7
2.3	Conditionnelles en Logo	7
3.1	Partie définition d'un fichier Flex	15
3.2	Options de Flex	15
3.3	Partie règles de Flex	15
3.4	Partie fonctions de Flex	15
3.5	Définition C++ en bison	15
3.6	Règles de grammaire en bison	16
3.7	Exemple du type des valeurs des jetons avant la version 0.3	16
3.8	Exemple du type des valeurs des jetons en version 1.0	17
4.1	Nommage lors de la création d'une tortue	21
4.2	Syntaxe des tables en Stibbons	22
5.1	Type des valeurs des jetons	25
5.2	Exemple de séquence d'instructions lors de la détection d'une couleur	26
5.3	Exemple de code Stibbons	27
5.4	Cas du IF en Bison	28
	src/interpreter/lexer.l+	53
	src/interpreter/parser.y+	58
	src/tests/test-agent.cpp	73
	doc/report/listings/sauvegarde.json	75

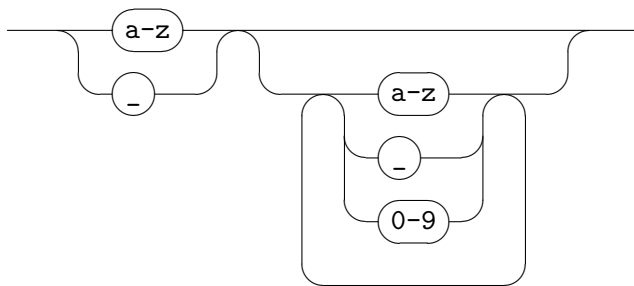
Annexe A

Documentation

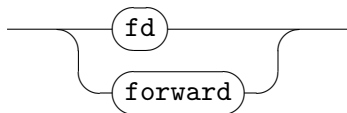
A.1 Syntaxe

A.1.1 Flex

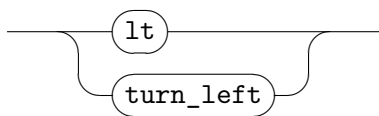
ID



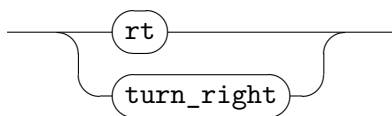
FD



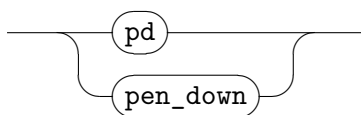
LT



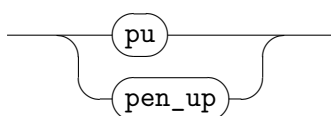
RT



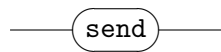
PD



PU



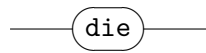
SEND



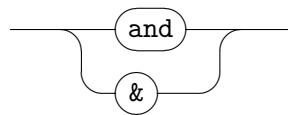
RECV



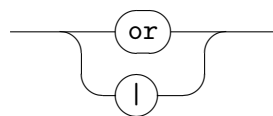
DIE



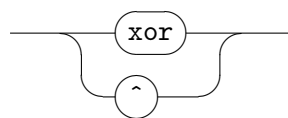
AND



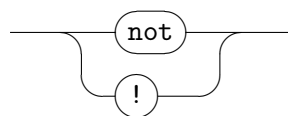
OR



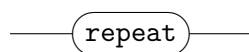
XOR



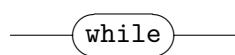
NOT



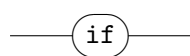
RPT



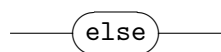
WHL



IF



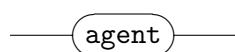
ELSE



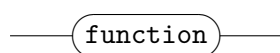
NEW



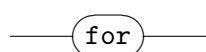
AGT



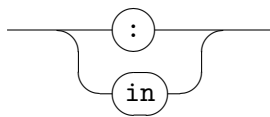
FCT



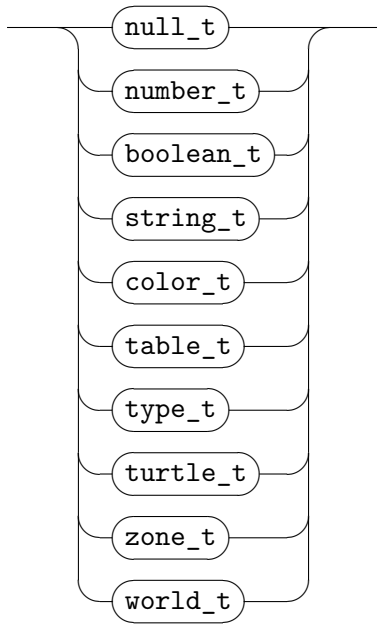
FOR



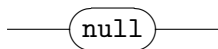
IN



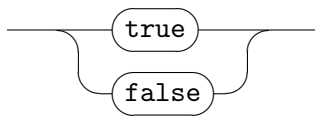
TYPE



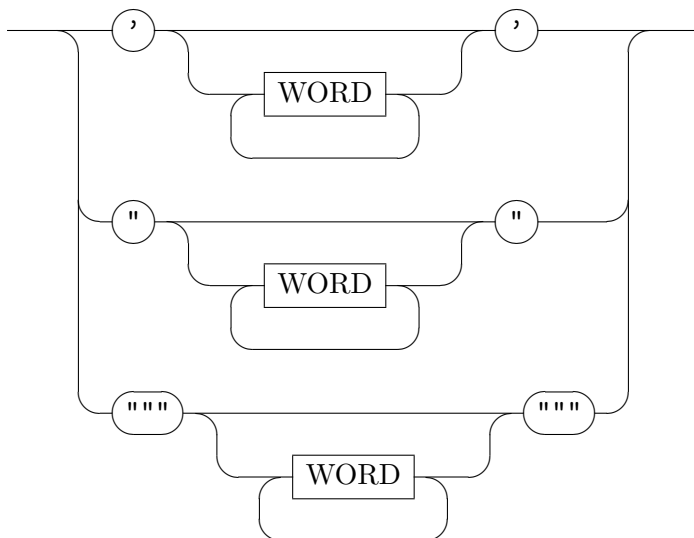
NIL



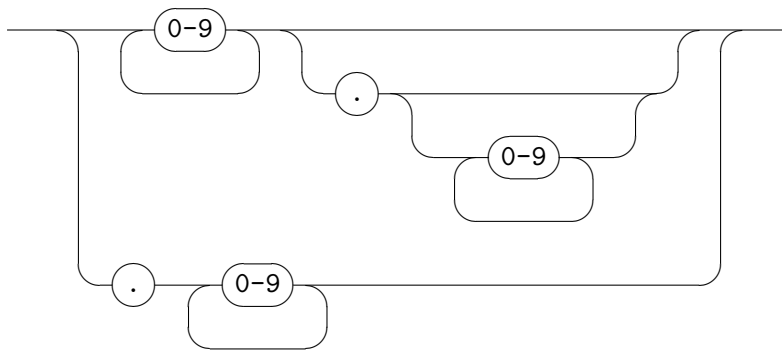
BOOLEAN



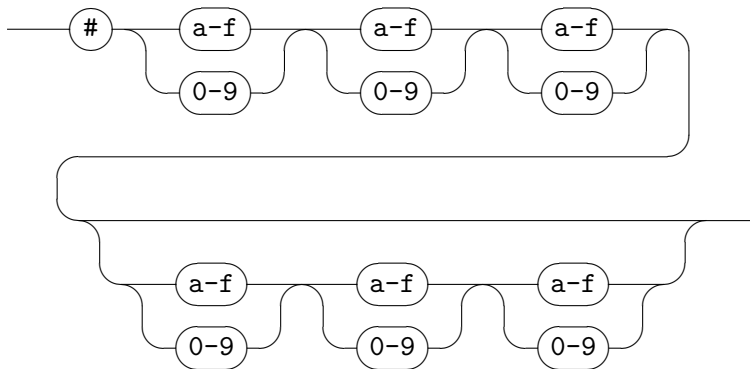
STRING



NUMBER

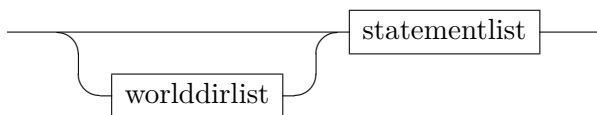


COLOR

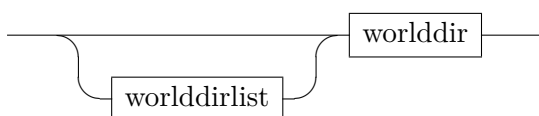


A.1.2 Bison

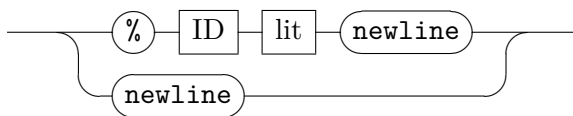
code



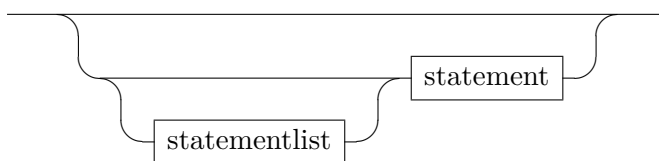
worlddirlist



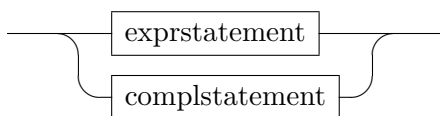
worlddir



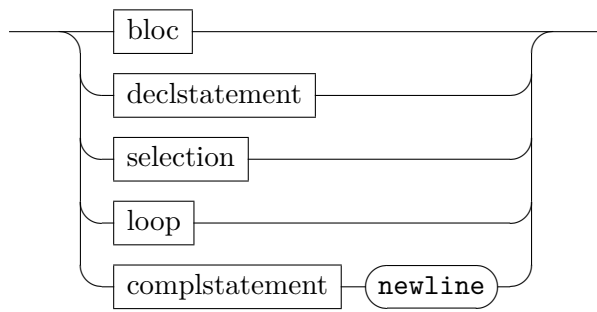
statementlist



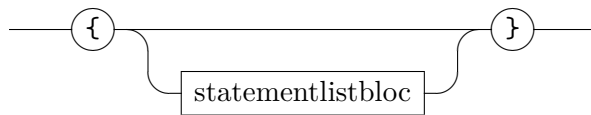
statement



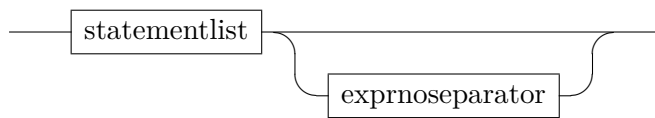
complstatement



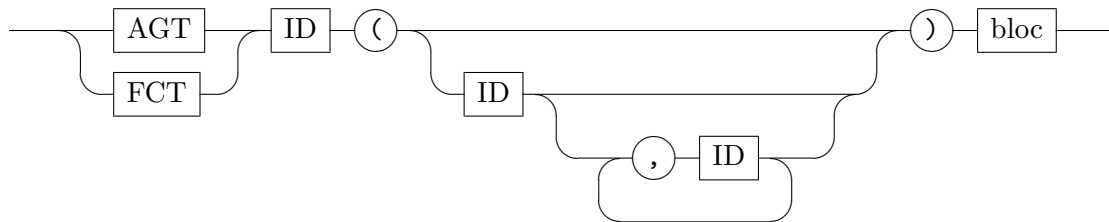
bloc



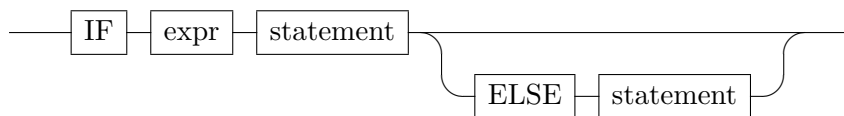
statementlistbloc



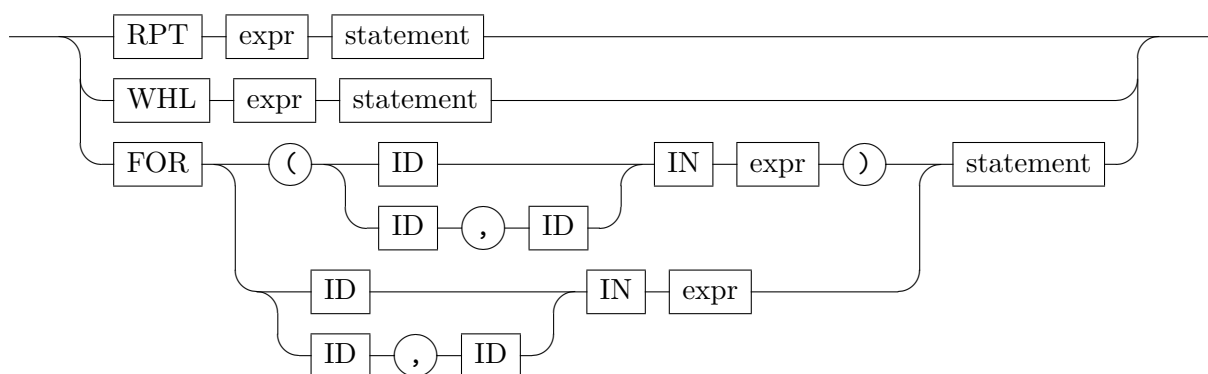
declstatement



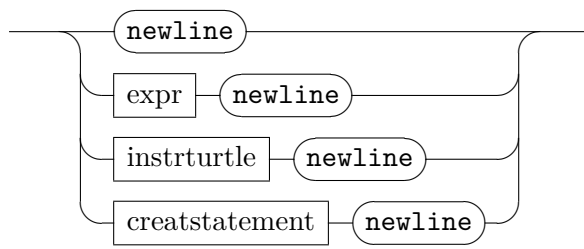
selection



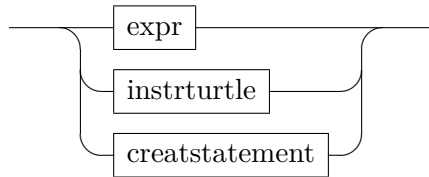
loop



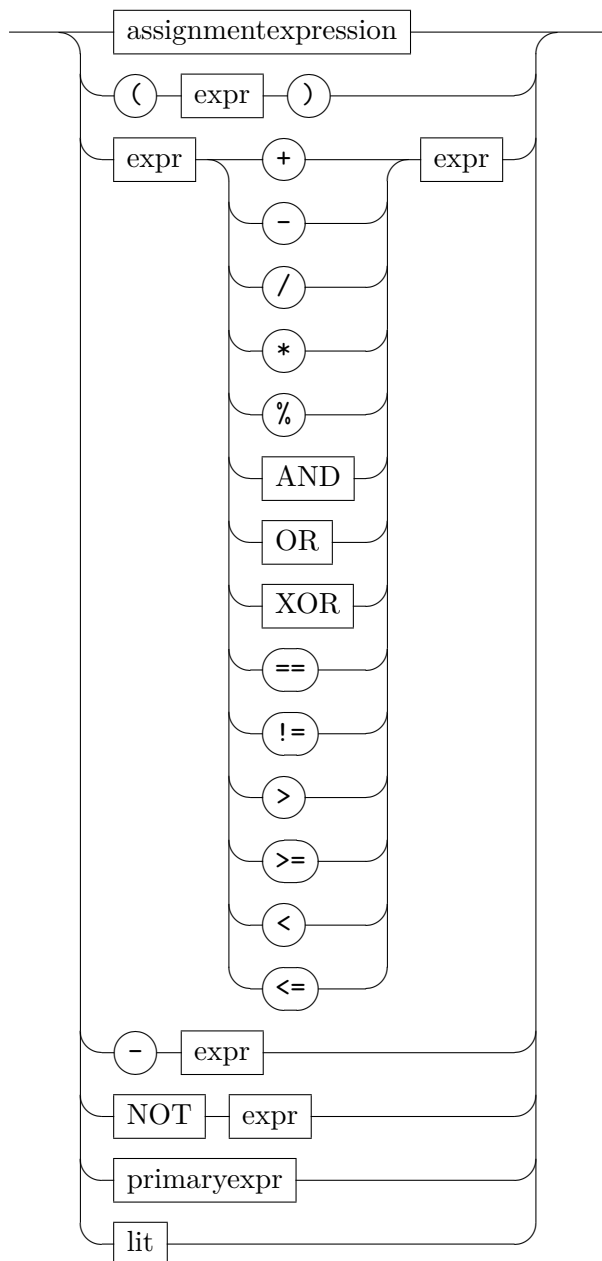
exprstatement



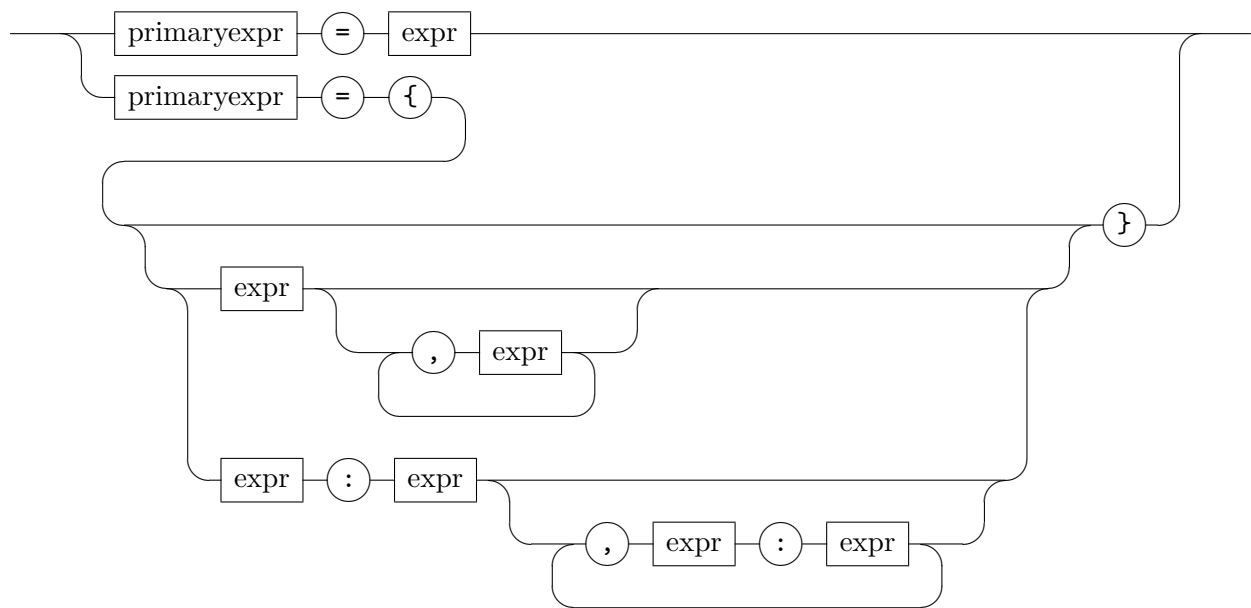
exprnoseparator



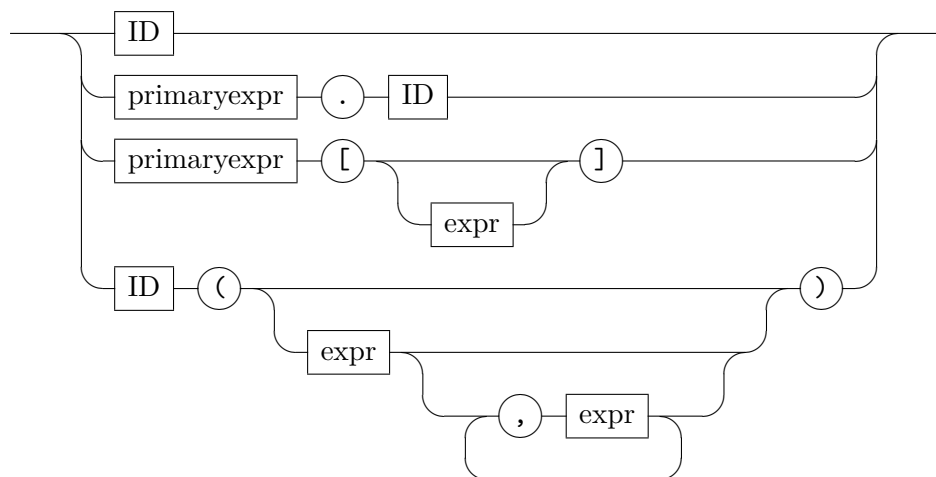
expr



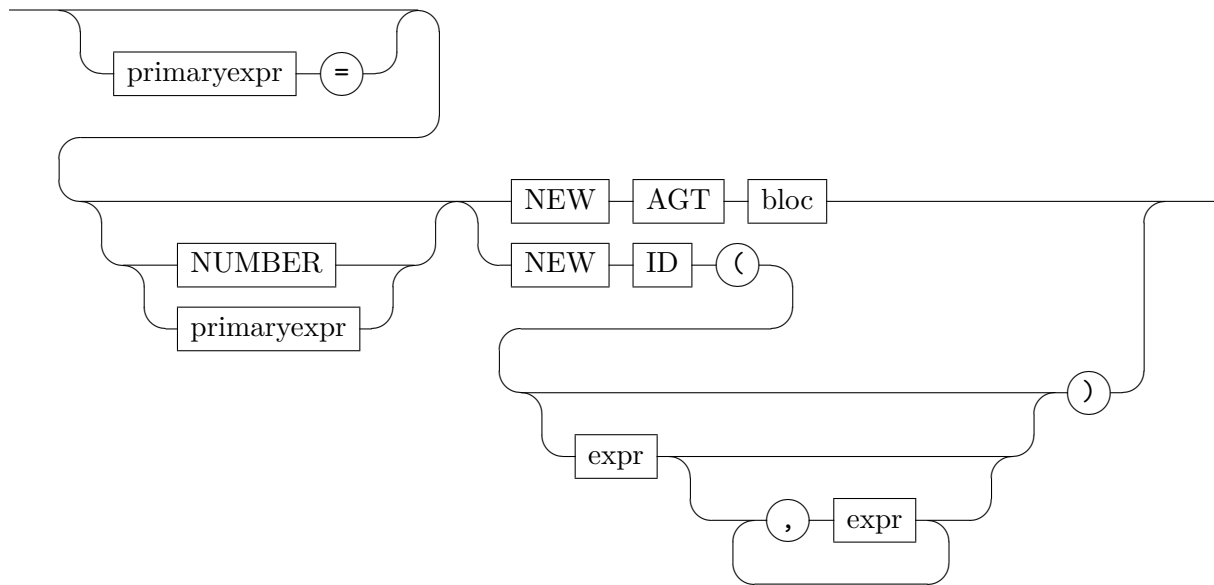
assignmentexpression



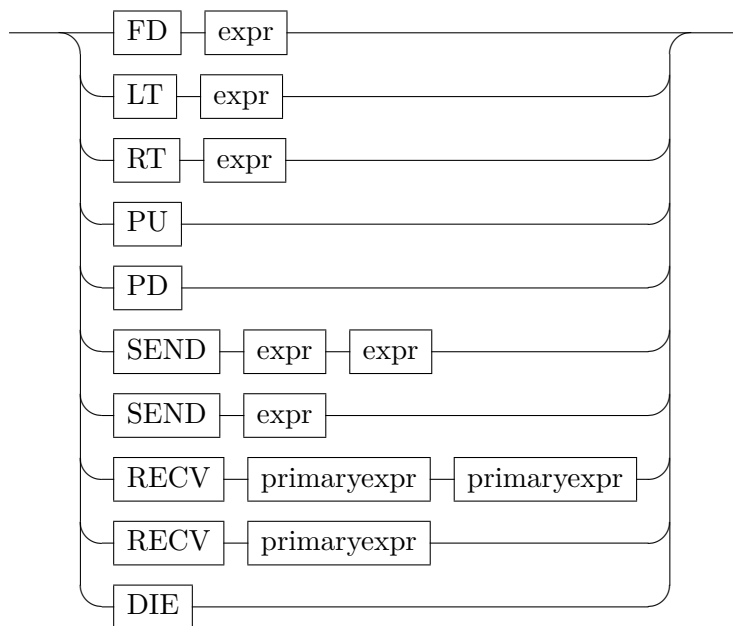
primaryexpr



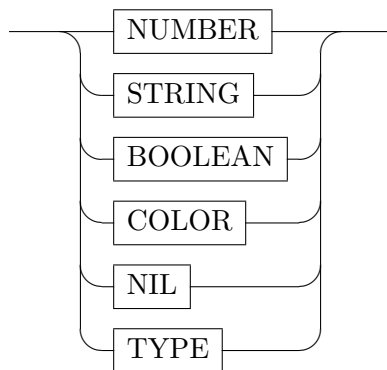
creatstatement



instrturtle



lit



A.2 Propriétés standard

Les agents possèdent certaines propriétés par défaut. Ces propriétés varient selon leur type, et peuvent être de simples valeurs en lecture seule, des fonctions en lecture seule, ou encore des propriétés spéciales ayant une sémantique particulière et requérant un type précis.

A.2.1 Attributs communs à tous les agents

black → #000000

Le noir.

Attribut en lecture seule.

white → #ffffff

Le blanc.

Attribut en lecture seule.

grey → #7f7f7f

Un gris moyen.

Attribut en lecture seule.

red → #ff0000

Le rouge.

Attribut en lecture seule.

green → #00ff00

Le vert.

Attribut en lecture seule.

blue → #0000ff

Le bleu.

Attribut en lecture seule.

yellow → #ffff00

Le jaune.

Attribut en lecture seule.

cyan → #00ffff

Le cyan.

Attribut en lecture seule.

magenta → #ff00ff

Le magenta.

Attribut en lecture seule.

A.2.2 Fonctions communes à tous les agents

print (value) → null

Imprime une valeur sur la sortie standard.

value La valeur à imprimer

Retourne null

println (value) → null

Imprime une valeur dans une nouvelle ligne sur la sortie standard.

value La valeur à imprimer

Retourne null

rand () → number

Retourne un entier positif au hasard.

Retourne un entier positif au hazard

random (min, max) → number

Retourne un entier positif au hasard compris entre les bornes indiquées.

min La valeur borne minimale inclusive

max La valeur borne maximale exclusive

Retourne un entier positif au hazard compris entre les bornes indiquées

type_of (value) → type

Retourne le type d'une valeur.

value La valeur dont on veut obtenir le type

Retourne le type de la valeur

size (table) → number

Retourne le nombre d'éléments contenus dans une table.

table La table dont on veut connaître le nombre d'éléments

Retourne le nombre d'éléments contenus dans la table

A.2.3 Attributs du monde

max_x → number

L'abscisse maximale du monde.

Attribut en lecture seule.

max_y → number

L'ordonnée maximale du monde.

Attribut en lecture seule.

A.2.4 Fonctions du monde

ask_zones (function) → null

Exécute une fonction sur chaque zone.

function La fonction à exécuter sur chaque zone

Retourne null

A.2.5 Attributs des tortues

color → #000000

La couleur de la tortue.

Requiert une valeur de type color.

parent → agent

L'agent qui a créé la tortue.

Attribut en lecture seule.

pos_x → number

L'abscisse de la position de la tortue.

Requiert une valeur de type number.

pos_y → number

L'ordonnée de la position de la tortue.

Requiert une valeur de type number.

pos_angle → number

L'angle de la tortue.

Requiert une valeur de type number.

world → world

Le monde.

Attribut en lecture seule.

zone → zone

La zone dans laquelle est la tortue.

Attribut en lecture seule.

A.2.6 Fonctions des tortues

distance_to (turtle) → number

Retourne la distance la plus courte vers une autre tortue.

turtle La tortue vers laquelle obtenir la distance

Retourne null

face (turtle) → null

Tourne la tortue pour qu'elle fasse face à une autre par le chemin le plus court.

turtle La tortue à laquelle faire face

Retourne null

in_radius (distance) → table

Retourne l'ensemble de tortues dans le rayon donné autour de la tortue.

distance Le rayon autour de la tortue à sonder

Retourne Une table contenant les tortues dans le rayon

inbox () → number

Retourne le nombre de message non lus.

Retourne Le nombre de messages non lus

teleport (x, y, angle) → null

Téléporte une tortue à une certaine coordonnée et avec un certain angle.

x L'abscisse où téléporter la tortue

y L'ordonnée où téléporter la tortue

angle L'angle à donner à la tortue

Retourne null

A.2.7 Attributs des zones

color → #fffff

La couleur de la zone.

Requiert une valeur de type color.

parent → agent

L'agent qui a créé la zone.

Attribut en lecture seule.

world → world

Le monde.

Attribut en lecture seule.

Annexe B

Tutoriel

B.1 Tutoriel

B.1.1 Salut, monde !

Commençons par imprimer du texte.

```
| println("Salut , monde !")
```

Ici l'agent par défaut, le monde, appelle la fonction `println` avec pour paramètre la chaîne de caractères "Salut, monde!", ce qui a pour effet d'imprimer ce texte dans une nouvelle ligne sur la sortie standard.

B.1.2 Les premiers agents

Créons maintenant des agents.

```
| new agent {  
    println("Salut , humain !")  
}
```

Le monde crée un nouvel agent mobile, une tortue, qui apparaîtra alors dans le monde et exécutera le code passé entre accolades.

Il est possible de créer plusieurs tortues exécutant le même code en spécifiant leur nombre.

```
| 5 new agent {  
    println("Salut , humain !")  
}
```

Ainsi, cinq tortues sont créées et chacune d'elles imprime "Salut, humain!".

B.1.3 Dessiner un carré

Les tortues peuvent se déplacer sur le monde en avançant et en tournant à gauche ou à droite. Elles ont également un stylo qu'elles peuvent abaisser ou relever afin de tracer des lignes sur le monde.

```
| new agent {  
    pd  
    fd 50  
    rt 90  
    fd 50  
    rt 90  
    fd 50
```

```

    rt 90
    fd 50
    println("Voici un beau carré !")
}

```

Ici, `pd` demande à la tortue d'abaisser son stylo (pen down), `fd` demande à la tortue d'avancer (forward) d'une certaine distance, et `rt` demande à la tortue de tourner d'un certain nombre de degrés.

B.1.4 Répéter

Afin d'éviter de se répéter, on peut demander à l'interprète de le faire un certain nombre de fois pour nous.

```

new agent {
    pd
    repeat 4 {
        fd 50
        rt 90
    }
    println("Voici qui est mieux. =)")
}

```

B.1.5 Boucler

Il est également possible de boucler tant qu'une condition est vraie.

```

new agent {
    println("Je vais faire ma ronde.")
    while true {
        fd 50
        rt 90
    }
}

```

B.1.6 Agents typés

Il est possible de définir un type d'agent sans en créer, afin d'en créer plus tard.

```

agent personne (nom) {
    println("Je m'appelle " + nom + ".")
}

new personne("Mathieu")
new personne("Michel")

```

Ici, le type d'agent `personne` a été défini. Un type d'agent peut prendre des paramètres exactement de la même manière qu'une fonction.

Ainsi on a pu créer deux tortues de type `personne`, chacune ayant son propre nom.

B.1.7 Fonctions

Il est possible de définir des fonctions. Les fonctions sont définies dans l'espace de nom des propriétés de l'agent.

```
%x_border bounce
%y_border bounce

agent fourmi () {
    function gigoter () {
        rt rand() % 60
        lt rand() % 60
        fd 1
    }

    while true {
        gigoter()
    }
}

new fourmi()
```

Les deux premières lignes seront expliquées un peu plus tard.

On définit ici la fonction `gigoter` pour les agents de type `fourmi`, qui est utilisée un peu plus bas dans le code.

B.1.8 Couleurs

Les tortues ont une couleur qui peut être modifiée. C'est également cette couleur qu'elles utilisent pour dessiner sur le monde.

```
%x_border bounce
%y_border bounce

agent fourmi (couleur) {
    function gigoter () {
        rt rand() % 60
        lt rand() % 60
        fd 1
    }

    color = couleur
    pd

    while true {
        gigoter()
    }
}

new fourmi(red)
new fourmi(blue)
```

B.1.9 Propriétés d'autres agents et parent

Il est possible d'accéder aux propriétés d'autres agents via l'opérateur « `.` ». De plus, il est possible d'accéder à certains agents via des propriétés spéciales, telles que `parent` pour obtenir le parent de l'agent actuel, `world` pour obtenir le monde et `zone` pour obtenir la zone sur laquelle se trouve une tortue.

Ces deux derniers types d'agents seront présentés un peu plus tard.

```
%x_border bounce
%y_border bounce

color = black

agent fourmi (enfants) {
    color = parent.color + #444

    if (enfants > 0) {
        2 new fourmi (enfants - 1)
    }

    function gigoter () {
        rt rand() % 60
        lt rand() % 60
        fd 1
    }

    while true {
        gigoter()
    }
}

new fourmi (2)
```

Dans cet exemple, chaque fourmi définit sa couleur en fonction de celle de son parent en l'obtenant via `parent.color`.

B.1.10 Le monde

Le monde est un agent très particulier : il est unique, immobile, possède une taille, et c'est dans son contexte qu'est exécuté le corps principal d'un programme Stibbons. Ça en fait de fait l'ancêtre commun à tous les autres agents.

```
5 new agent {
    teleport(rand() % world.max_x, rand() % world.max_y, rand())
}
```

Cet exemple crée 5 nouveaux agents et, grâce aux fonctions `teleport` et `rand` et aux propriétés spéciales du monde `max_x` et `max_y`, positionne chaque agent une position et un angle au hasard à l'intérieur des bords du monde.

B.1.11 Les zones

Le monde est constitué de zones qui sont elles aussi des agents. Les zones sont immobiles, colorées, et ont le monde pour parent.

```
%x_border bounce
%y_border bounce

agent fourmi (couleur) {
    teleport(rand() % world.max_x, rand() % world.max_y, rand())
}
```



```

function gigoter () {
    rt rand() % 60
    lt rand() % 60
    fd 1
}

color = couleur - #444
couleur_zone = couleur + #888

while (true) {
    gigoter()
    zone.color = couleur_zone
}
}

new fourmi(red)
new fourmi(green)
new fourmi(blue)
new fourmi(yellow)
new fourmi(cyan)
new fourmi(magenta)

```

Ici, les fourmis changent la couleur des zones sur lesquelles elles passent.

B.1.12 Directives de monde

Dans cette section seront enfin expliquées les mystérieuses instructions `%x_border bounce` et `%y_border bounce`.

Le monde peut être paramétré au chargement du programme, pour cela on utilise des directives de monde qui doivent toutes être placées en tout début du programme.

Les directives `%world_width` et `%world_height` permettent de définir le nombre de zones constituant le monde en largeur et en hauteur, et doivent être suivi du nombre souhaité. Les directives `%zone_width` et `%zone_height` permettent de définir la largeur et la hauteur des zones constituant le monde, et doivent être suivi de la taille souhaitée. Les directives `%x_border` et `%y_border` permettent de définir le comportement des tortues lorsqu'elles franchissent un bord du monde. Elles doivent être suivies de `none` pour laisser les tortues sortir du monde, de `bounce` pour faire rebondir les tortues contre les bords ou de `wrap` pour reboucler le monde sur lui même par l'axe en question.

```

%world_width 100 // Le monde aura 100 zones en largeur
%world_height 100 // Le monde aura 100 zones en hauteur
%zone_width 5 // Les zones feront 5 unités de large
%zone_height 5 // Les zones feront 5 unités de haut
%x_border wrap // Le monde rebouclera sur lui même sur les bords
                verticaux
%y_border bounce // Le monde sera solide sur les bords horizontaux

```

B.1.13 Messages

Les tortues peuvent s'envoyer des messages. Les messages peuvent être envoyés à un destinataire précis, à un ensemble de destinataires ou à toutes les tortues.

```

destinataire = new agent {
    recv message

```

```

    println("Quelqu'un m'a dit : " + message)
}

```

```

new agent {
    send "Coucou !" world.destinataire
}

```

```

5 new agent {
    recv message

    println("Quelqu'un m'a dit : " + message)
}

```

```

new agent {
    send "Oyez, agents !"
}

```

S'il n'y a aucun message dans sa boîte à messages, une tortue demandant la lecture d'un message sera bloquée jusqu'à réception d'un message à lire. Pour éviter un blocage, il est possible de vérifier le nombre de messages présents dans la boîte.

```

new agent {
    new agent {
        send "Bonjour, parent !" parent
    }

    while true {
        if (inbox() > 0) {
            recv message

            println("Quelqu'un m'a dit : " + message)
        }

        rt 1
        fd 1
    }
}

```

Annexe C

Résumés des réunions

C.1 27 janvier 2015

Voici le résumé de la première réunion, le 27 janvier.

Tout d'abord, petite précision et chose à faire rapidement :

- trouver un article à lire sur le multi-agents
- le rapport sera fait en Latex
- il faut créer un dépôt GIT du Sif pour mettre le travail et le rapport

Résumé du projet :

Nous allons développer un système multi-agent sur une même machine, pas de réseau.

Nous avons d'abord parler de faire un ordonnanceur de tâche pour exécuter les agents un à un, comme en NetLOGO, puis nous avons plutôt opter pour l'exécution en parallélisme des tortues, car faire un ordonnanceur causerai des difficultés, (comme comment récupérer la main sur les tortues(utilisation de alarm, mais on ne sait pas qui, ect),et nous préférons utiliser les mécanismes UNIX existant. Nous utiliserons des threads pour programmer les processus tortues plutôt que les forks car ils sont plus modernes, et nous permettront l'usage de moyens de communication comme la mémoire partagée (vs les files de msg avec fork), une meilleure performance, et pour la synchronisation il y aura les mutexs (vs les sémaphores avec fork).

Chaque thread devra interpréter son code.

On aura un thread_main qui aura ses instructions, et la main. Il sera le thread pré-existant, et il y aura des thread_turtles pour les tortues. Nous ferons un MVC, et utiliserons Qt pour l'interface, et C++.

Nous allons procéder de manière incrémentale, avec la méthode AGILE. C'est à dire que nous commencerons par avoir un noyau : une tortue qui s'affiche sur l'interface graphique 2D et qui peut executer 3 instructions simples. Pour commencer, le plan aura une taille fixe. Nous améliorerons l'interpréteur au fur et à mesure et nous ferons des intégrations successives des fonctionnalités.

A faire :

- Quels sont ses moyens de communication des agents ?
- Réfléchir si 1 agent = 1 thread ? thread séquentiel ? système de jeton ?
- Les agents peuvent-ils communiquer avec les patches ? Avec qui peut-on communiquer ?

C.2 3 février 2015

Lors de cette réunion, nous avons principalement discuté du projet, et nous nous sommes mis d'accord sur quelques points.

Tout d'abord, nous utiliserons un thread par tortue, car cela ne devrait pas poser de problème de performance, et cela permettra l'exécution en parallèle des tortues, ce qui change de NetLogo.

Nous avons parlé de mettre une option pour exécuter pas à pas ces threads.

A propos du vocabulaire du projet, nous avons choisi d'appeler les patchs de NetLogo des zones, et pour la tortue d'origine nous la nommons le dieu-tortue.

Pour la communication, elle sera autorisée entre tortues et zones, et éventuellement limitée par la distance.

Pour la partie technique, le main devrait gérer les zones, et la communication se fera avec des files de messages. Nous avons également parlé d'utiliser des classes anonymes pour remplacer la fonction go dans les tortues.

C'est ce qui est ressorti de nos discussions, mais ces décisions ne sont pas encore définitive.

C.3 10 février 2015

Voici la liste des tâches à faire pour la semaine prochaine (si assignées).

C.3.1 Analyse de l'existant

- Logo → Florian
- NetLogo → Clément
- StarLogo → Clément

C.3.2 Analyse des outils

- Gestion de projet :
 - producteev
 - openproj
 - git
 - make
- Tests unitaires → Florian
- C++11 ou supérieur
 - Threads standard C++ → Florian
 - gdb
- Flex, Bison → Julia
- Qt → Adrien
 - Qt en général
 - Dessiner avec Qt
- LaTeX (et Beamer)

C.4 24 février 2015

Lors de cette réunion nous avons tout d'abord fait un point sur le backlog et les tâches que nous avons choisi pour le premier sprint :

- interprète simple, qui ne comprend que des instructions basiques ;
- une seule tortue est gérée ;
- pas d'éditeur intégré dans un premier temps mais seulement lecture de code dans un fichier externe.

Nous avons également précisé que le code analysé serait lu en intégralité et interprété, et non pas interprété ligne à ligne, tout du moins dans un premier temps. Ainsi, nous passerons par une phase de pré-compilation afin d'analyser le code source stibbons. Nous aurons donc un schéma du type : code source → tokenizer → analyse et interprétation

Nous définirons dans un premier temps un langage élémentaire, contenant les affectations ainsi que les instructions de bases pour les tortues (forward, turn-right, etc.) et augmenterons

la complexité du langage dans le temps. Il faudra ainsi définir très rapidement une grammaire, l'idéal étant pour la fin de la semaine. Cette tâche est assignée à Florian et Clément.

Pendant ce temps-là, Adrien et Julia sont chargés de définir le modèle. On utilisera pour cela le langage UML, ainsi que le logiciel Umbrello qui permet la génération de code.

Nous avons également abordé la question de la gestion de la bibliographie. Nous utiliserons bibtex dans le rapport pour celle-ci.

C.5 16 mars 2015

Bilan du premier sprint, par rapport à ce qui était prévu, nous notons les différences suivantes :

- le chargement et l'importation n'est pas complet, on doit passer par le terminal ;
- l'interprétation d'un agent se fait, mais limité.

Pour ce qui est de l'estimation des temps des tâches, nous étions globalement au-dessus. Nous avons ajouté quatre nouvelles tâches au backlog :

- ajout de fonctions ;
- ajout des variables ;
- ajout des boucles ;
- ajout des conditionnelles.

Pour la version 0.2 nous avons choisis les tâches suivantes :

- l'utilisateur utilise une variable ;
- l'utilisateur utilise une fonction ;
- les tortues fonctionnent en parallèle ;
- ajout de l'utilisation de breed.

Nous avons défini les tests pour ces tâches.

Nous avons également effectué les estimations de la durée de chaque tâche en écrivant chacun le temps minimum et le temps maximum estimé, puis nous avons fait la moyenne de ces temps, en discutant des raisons des résultats. Nous estimons donc le deuxième sprint à deux semaines, à raison de cinq après-midi de cinq heures par personne (environ 96 heures au total).

C.6 17 mars 2015

Nous avons effectué une réunion avec notre encadrant suite à la fin de notre premier sprint. Certains points ont été soulevés tels que la communication inter-agents et le moyen de l'implémenter mais cela concerne la version 0.3, nous devons donc commencer à y réfléchir sans le réaliser. Le multi-agents de la version 0.2 sera réalisé à l'aide de threads, avec un thread pour chaque tortue.

Le diagramme UML a également été ré-étudié, nous avons donc remarqué certaines différences avec l'implémentation réalisée au cours de la version 0.1, notamment concernant les classes :

- Value ;
- Type ;
- Colored ;
- Boolean ;
- Color ;
- Les classes liées à l'interpréteur (Tree, Interpreter).

Certaines conceptions ont été revues :

- La classe Shape devrait être reliée à une Breed, pas à une Tortue : la Breed définira le comportement d'une tortue ;
- La classe Breed : renommage en TurtleClass ;
- Les agrégations et compositions entre les classes sont à rectifier.

- Pour terminer, deux pistes concernant la gestion de l'arbre syntaxique ont été évoqué :
- Réaliser un arbre de dérivation avec chaque frère droit qui est un instruction ;
 - Un arbre abstrait en UML.

C.7 7 avril 2015

Nous avons fini le second sprint. Nous avons surtout ajouté des choses dans le modèle et dans l'interpréteur : il s'agissait de la prise en charge du multi-threading et des variables.

Coté modèle, on a ajouté les classes `Function`, `Breed`, adapté la classe `Turtle`, ajouté des mutexs dans chaque classe qui le demandait. Nous avons également rajouté un système de "parenté", qui permet à une tortue de connaître son parent et d'avoir accès à ses propriétés.

Coté interprète, il a fallu mettre en place les threads, un à chaque "new agent", et les fonctions. Lors de cette dernière réunion, nous avons discuté de comment afficher les résultats de simulation de notre programme : diagramme, variables tracées avec un journal... Il faudra choisir une simulation de NetLogo, la tester et comparer les résultats des deux applications.

Pour le moment, nous devons redémarrer le programme pour lancer un nouveau fichier. Il faudra utiliser `yywrap`, qui chaîne les fichiers.

Pour la taille du monde, nous pensons à faire une taille par défaut si l'utilisateur n'en choisit pas, et lui donné la possibilité d'en choisir une en début de fichier grâce à une syntaxe différente du stibbons.

Annexe D

Listing

D.1 Flex

```
/*
 * This file is part of Stibbons.
 *
 * Stibbons is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as
 * published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Stibbons is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License
 * along with Stibbons. If not, see <http://www.gnu.org/licenses/>.
 */

%{
#include "flex-scanner.h"
#define YY_NO_UNISTD_H
%}

%option c++
%option nodefault
%option yyclass="FlexScanner"
%option yywrap nounput

%option case-insensitive

/* Define tokens */
%{
#include <sstream>
#include <cstdlib>
#include "y.tab.h"
```

```

#include "syntax-exception.h"
#include "../model/nil.h"
#include "../model/number.h"
#include "../model/string.h"
#include "../model/boolean.h"
#include "../model/color.h"
#include "../model/type-value.h"

#define YY_USER_ACTION loc->columns(yyval);
#define yyterminate() {}

size_t countNewLine(char* str) {
    auto count = 0;
    for(auto i=strlen(str)-1;i>=0;i--) {
        if(str[i] == '\n')
            ++count;
        else
            break;
    }

    return count;
}

int yyFlexLexer::yywrap() {
    return 1;
}
%}

id [_a-z][_a-z0-9]*

%x comment
%x spl_quote
%x dbl_quote
%x tpl_quote
%s for_state
%x end

/* Rules */
%%

std::string tmp;
%{
    loc->step();
}%

<INITIAL><<EOF>> { BEGIN(end); return yy::parser::token_type('\n'); }
<end><<EOF>> { BEGIN(INITIAL); return 0; }

/* Creating instruction tokens */
true|false { if(yytext[0] == 't' || yytext[0] == 'T') { pyylval->v =
    make_shared<stibbons::Boolean>(true); } else { pyylval->v =
    make_shared<stibbons::Boolean>(false); } return yy::parser::token

```



```

::BOOLEAN;}

null {return yy::parser::token::NIL;}

fd|forward {return yy::parser::token::FD;}

lt|turn_left {return yy::parser::token::LT;}

rt|turn_right {return yy::parser::token::RT;}

pd|pen_down {return yy::parser::token::PD;}

pu|pen_up {return yy::parser::token::PU;}

send {return yy::parser::token::SEND;}

recv {return yy::parser::token::RECV;}

die {return yy::parser::token::DIE;}

/*Creating comparison operations tokens*/
"==" {return yy::parser::token::EQ;}

"!=" {return yy::parser::token::NEQ;}

">" {return yy::parser::token::GT;}

">=" {return yy::parser::token::GEQ;}

"<" {return yy::parser::token::LS;}

"<=" {return yy::parser::token::LEQ;}

and|& {return yy::parser::token::AND;}

or|\| {return yy::parser::token::OR;}

xor|^ {return yy::parser::token::XOR;}

not|! {return yy::parser::token::NOT;}

/*Creating loops (except 'for') and conditional tokens*/
repeat {return yy::parser::token::RPT;}

while {return yy::parser::token::WHL;}

if {return yy::parser::token::IF;}

else {return yy::parser::token::ELSE;}

/*Creating agent and function tokens*/

```

```

new {return yy::parser::token::NEW;}

agent {return yy::parser::token::AGT;}

function {return yy::parser::token::FCT;}

/*Creating tokens type*/
null_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    NIL);
    return yy::parser::token::TYPE;}

number_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type
    ::NUMBER);
    return yy::parser::token::TYPE;}

boolean_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type
    ::BOOLEAN);
    return yy::parser::token::TYPE;}

string_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type
    ::STRING);
    return yy::parser::token::TYPE;}

color_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    COLOR);
    return yy::parser::token::TYPE;}

table_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    TABLE);
    return yy::parser::token::TYPE;}

type_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    TYPE);
    return yy::parser::token::TYPE;}

turtle_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type
    ::TURTLE);
    return yy::parser::token::TYPE;}

zone_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    ZONE);
    return yy::parser::token::TYPE;}

world_t {pyylval->v=make_shared<stibbons::TypeValue>(stibbons::Type::
    WORLD);
    return yy::parser::token::TYPE;}

/*Creation loop 'for' tokens*/
for { BEGIN(for_state); return yy::parser::token::FOR; }

<for_state>in {BEGIN(INITIAL); return yy::parser::token_type(':');}

```

```

<for_state>': ' {BEGIN(INITIAL); return yy::parser::token_type(':');}

/*Creating string tokens*/
\" \" \" { tmp = std::string(); BEGIN(tpl_quote); }

<tpl_quote>\" \" \" { BEGIN(INITIAL);
    pyylval->v=make_shared<stibbons::String>(tmp);
    return yy::parser::token::STRING;}
<tpl_quote>[^\" ]+ { tmp += yytext; }
<tpl_quote>\" [^\" ] { tmp += yytext; }
<tpl_quote>\" \" [^\" ] { tmp += yytext; }

\" { tmp = std::string(); BEGIN(dbl_quote); }

<dbl_quote>[^\\\"\\n\\"]+ { tmp += yytext; }
<dbl_quote>\\\" { tmp += yytext[1]; }
<dbl_quote>\\n { throw stibbons::SyntaxException("New line in a \"
    string\" ",loc->begin); }
<dbl_quote>\" { BEGIN(INITIAL);
    pyylval->v=make_shared<stibbons::String>(tmp);
    return yy::parser::token::STRING;}

\' { tmp = std::string(); BEGIN(spl_quote); }

<spl_quote>[^\\\"\\n\\'] + { tmp += yytext; }
<spl_quote>\\\' { tmp += yytext[1]; }
<spl_quote>\\n { throw stibbons::SyntaxException("New line in a \'
    string\' ",loc->begin); }
<spl_quote>\' { BEGIN(INITIAL);
    pyylval->v=make_shared<stibbons::String>(tmp);
    return yy::parser::token::STRING;}

<spl_quote,dbl_quote>\\\\\\\\ { tmp += yytext[1]; }
<spl_quote,dbl_quote>{
    \\n { tmp += '\\n'; }
    \\r { tmp += '\\r'; }
    \\t { tmp += '\\t'; }
    \\f { tmp += '\\f'; }
    \\0 { tmp += '\\0'; }
}

/*Creating number tokens*/
[0-9]+(\\. [0-9]*)?\\. [0-9]+ {std::istringstream iss(yytext); double n;
    iss >> n; pyylval->v=make_shared<stibbons::Number>(n); return yy
    ::parser::token::NUMBER;}

/*Creating color tokens*/
#([a-f0-9]{6}|[a-f0-9]{3}) {pyylval->v=make_shared<stibbons::Color>(
    yytext); return yy::parser::token::COLOR;}

/*Creating identifying variables tokens*/

```

```

{id} {pyylval->v=make_shared<stibbons::String>(yytext); return yy::
    parser::token::ID;}

/*Creating comment tokens*/
"//[^\n]*\n* { loc->lines(countNewLine(yytext)); loc->step(); return
    yy::parser::token_type('\n');}

"/*" {BEGIN(comment);}

<comment>[^\n]* {loc->lines(countNewLine(yytext)); loc->step();}
<comment>"*" + [^\n]*
<comment>"*" + "/" {BEGIN(INITIAL);}

[ \r\t]+ {loc->step();}

\n\n* { loc->lines(yyval); loc->step(); return yy::parser::
    token_type(yytext[0]);}

. {return yy::parser::token_type(yytext[0]);}

%%

/*
 * Editor modelines - http://www.wireshark.org/tools/modelines.html
 *
 * Local variables:
 * mode: c++
 * c-basic-offset: 4
 * tab-width: 4
 * indent-tabs-mode: t
 * truncate-lines: 1
 * End:
 *
 * vim: set ft=cpp ts=4 sw=4 sts=4
 */

```

D.2 Bison

```

/*
 * This file is part of Stibbons.

 * Stibbons is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as
 * published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.

 * Stibbons is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.

```

```

* You should have received a copy of the GNU Lesser General Public
   License
   * along with Stibbons. If not, see <http://www.gnu.org/licenses/>.
   */

%skeleton "lalr1.cc"
%defines
%locations
%parse-param { stibbons::FlexScanner &scanner }
%parse-param { stibbons::TreePtr t }
%parse-param { stibbons::TablePtr w }
%lex-param { stibbons::FlexScanner &scanner }

%code requires {
    namespace stibbons {
        class FlexScanner;
    }
#include "tree.h"
#include "../model/table.h"
#include "../model/string.h"
#include "../model/number.h"
#define YYSTYPE struct { stibbons::ValuePtr v; stibbons::TreePtr tr;
    int tok; }
    std::string toString(const int& tok);
}

%code {
#include "flex-scanner.h"
#include "syntax-exception.h"
    using namespace std;

    static int yylex(yy::parser::semantic_type* pyylval,
                    yy::parser::location_type* loc,
                    stibbons::FlexScanner &scanner);

    void yy::parser::error(const location& loc, string const& s) {
        throw stibbons::SyntaxException(s.c_str(), loc.begin);
    }

    static int yylex(yy::parser::semantic_type* pyylval,
                    yy::parser::location_type* loc,
                    stibbons::FlexScanner &scanner) {
        return scanner.yylex(pyylval, loc);
    }

    std::string toString(const int& tok) {
        switch(tok) {
            case yy::parser::token::SEQ:
                return "SEQ";
            case yy::parser::token::FD:

```

```

    return "FD" ;
case yy::parser::token::RT:
    return "RT" ;
case yy::parser::token::LT:
    return "LT" ;
case yy::parser::token::PU:
    return "PU" ;
case yy::parser::token::PD:
    return "PD" ;
case yy::parser::token::SEND:
    return "SEND" ;
case yy::parser::token::RECV:
    return "RECV" ;
case yy::parser::token::DIE:
    return "DIE" ;
case yy::parser::token::AND:
    return "AND" ;
case yy::parser::token::OR:
    return "OR" ;
case yy::parser::token::XOR:
    return "XOR" ;
case yy::parser::token::NOT:
    return "NOT" ;
case yy::parser::token::EQ:
    return "EQ" ;
case yy::parser::token::NEQ:
    return "NEQ" ;
case yy::parser::token::GT:
    return "GT" ;
case yy::parser::token::GEQ:
    return "GEQ" ;
case yy::parser::token::LS:
    return "LS" ;
case yy::parser::token::LEQ:
    return "LEQ" ;
case yy::parser::token::CALL:
    return "CALL" ;
case yy::parser::token::RPT:
    return "RPT" ;
case yy::parser::token::WHL:
    return "WHL" ;
case yy::parser::token::IF:
    return "IF" ;
case yy::parser::token::ELSE:
    return "ELSE" ;
case yy::parser::token::FCT:
    return "FCT" ;
case yy::parser::token::NEW:
    return "NEW" ;
case yy::parser::token::AGT:
    return "AGT" ;

```

```

    case yy::parser::token::NUMBER:
        return "NUMBER";
    case yy::parser::token::STRING:
        return "STRING";
    case yy::parser::token::COLOR:
        return "COLOR";
    case yy::parser::token::BOOLEAN:
        return "BOOLEAN";
    case yy::parser::token::NIL:
        return "NIL";
    case yy::parser::token::ID:
        return "ID";
    case yy::parser::token::TABLE:
        return "TABLE";
    case yy::parser::token::PAIR:
        return "PAIR";
    case yy::parser::token::UNARYMINUS:
        return "-";
    case yy::parser::token::FOR:
        return "FOR";
    default:
        return std::string(1,static_cast<char>(tok));
}
}
}

```

```

%token SEQ 0          "sequence "
%token FD             "forward "
%token LT             "turn-left "
%token RT             "turn-right "
%token PU             "pen-up "
%token PD             "pen-down "
%token SEND           "send "
%token RECV           "recv "
%token DIE            "die "
%token AND             "and "
%token OR             "or "
%token XOR            "xor "
%token NOT            "not "
%token EQ             "=="
%token NEQ            "!="
%token GT             "> "
%token GEQ            ">="
%token LS             "< "
%token LEQ            "<="
%token RPT            "repeat "
%token WHL            "while "
%token IF             "if "
%token ELSE           "else "
%token FCT            "function "
%token NEW            "new "

```

```

%token AGT          "agent "
%token CALL         "call "
%token TABLE       "table "
%token PAIR          "pair "
%token UNARYMINUS    "-"
%token FOR           "for "
%token IN            "in "
%token ATT_ID
%token TAB_ID
%token <v> NUMBER
%token <v> STRING
%token <v> COLOR
%token <v> BOOLEAN
%token <v> NIL
%token <v> TYPE
%token <v> ID
%type <tr> nb_agt
%type <tr> statement_list
%type <tr> statement_list_bloc
%type <tr> statement
%type <tr> instr_turtle
%type <tr> compl_statement
%type <tr> bloc
%type <tr> expr
%type <tr> lit
%type <tr> assignment_expression
%type <tr> expr_statement
%type <tr> expr_no_separator
%type <tr> decl_statement
%type <tr> creat_statement
%type <tr> primary_expr
%type <tr> loop
%type <tr> for_expr
%type <tr> selection
%type <tr> decl_list
%type <tr> id_list
%type <tr> initializer_list
%type <tr> table_list
%type <tr> expr_list
%type <tr> pair_list
%type <tr> pair
%type <tok> binary_operator
%type <tok> decl_id

%right '='
%left AND OR XOR
%left EQ NEQ
%left GT GEQ LS LEQ
%left '+' '-'
%left '*' '/' '%'
%right MOINSUNAIRE NOT

```



```

%nonassoc '{' '}' '(' ')',
%%
code : world_dir_list statement_list { t->addChild($2); }
| statement_list { t->addChild($1); }
;

//Storage of the world's directives

world_dir_list : world_dir {}
| world_dir_list world_dir {}
;

world_dir : '%' ID lit '\n'
{
    auto val = std::get<1>($3->getNode());
    w->setValue(dynamic_pointer_cast<stibbons::String>($2)->getValue(),
        val);
}
| '%' ID ID '\n'
{
    w->setValue(dynamic_pointer_cast<stibbons::String>($2)->getValue(),
        $3);
}
| '\n' {}
;

// General languages struct
// Storage of tokens and contents in a tree

//Storage of statement and bloc
statement_list : statement
{
    $$ = $1;
}
| statement_list statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(0,nullptr);
    try {
        t1->mergeTree($1);
    }
    catch(std::exception& e) {
        t1->addChild($1);
    }
    t1->addChild($2);
    $$ = t1;
}
| { $$ = nullptr; }
;

statement : expr_statement { $$ = $1; }
| compl_statement { $$ = $1; }

```

```

;

compl_statement : bloc { $$ = $1; }
| decl_statement { $$ = $1; }
| selection { $$ = $1; }
| loop { $$ = $1; }
| compl_statement '\n' { $$ = $1; }
;

bloc : '{' '}' { $$ = nullptr; }
| '{' statement_list_bloc '}' { $$ = $2; }
;

statement_list_bloc : statement_list { $$ = $1; }
| statement_list expr_no_separator {
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(0, nullptr);
    try {
        t1->mergeTree($1);
    }
    catch(std::exception& e) {
        t1->addChild($1);
    }
    t1->addChild($2);
    $$ = t1;
}
;

//Storage of declaration of object (id, variables, list, ...)
decl_statement : decl_id ID decl_list bloc {
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>($1, $2);
    t1->setPosition({@1.begin.line, @1.begin.column});
    t1->addChild($4);
    t1->appendChildren($3);
    $$ = t1;
}
;

decl_id : AGT { $$ = yy::parser::token::AGT; }
| FCT { $$ = yy::parser::token::FCT; }
;

decl_list : '(' ')' { $$ = nullptr; }
| '(' id_list ')' { $$ = $2; }
;

id_list : ID
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::ID, nullptr);
    t1->addChild(make_shared<stibbons::Tree>(yy::parser::token::ID, $1))
    ;
}
;

```

```

    $$ = t1;
}
| id_list ' , ' ID
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::ID, nullptr);
    t1->appendChildren($1);
    t1->addChild(make_shared<stibbons::Tree>(yy::parser::token::ID,$3))
        ;
    $$ = t1;
};

//Storage of conditionnal expression
selection : IF expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}
| IF expr statement ELSE statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::IF, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->addChild($5);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
};

//Storage of loop expression
loop : RPT expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::RPT, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}
| WHL expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::WHL, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}

```

```

}
| FOR for_expr statement
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::FOR, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}
;

for_expr : '(' for_expr ')' { $$ = $2; }
| ID ':' expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::FOR,$1);
    t1->addChild($3);
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}
| ID ',' ID ':' expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::FOR,$3);
    t1->addChild($5);
    t1->addChild(make_shared<stibbons::Tree>(yy::parser::token::ID,$1))
        ;
    t1->setPosition({@1.begin.line ,@1.begin.column});
    $$ = t1;
}
;

//Storage of expression
expr_statement : '\n' { $$ = nullptr; }
| expr '\n' { $$ = $1; }
| instr_turtle '\n' { $$ = $1; }
| creat_statement '\n' { $$ = $1; }
;

expr_no_separator : expr { $$ = $1; }
| instr_turtle { $$ = $1; }
| creat_statement { $$ = $1; }
;

expr : assignment_expression { $$ = $1; }
| '(' expr ')' { $$ = $2; }
| expr binary_operator expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>($2, nullptr);
    t1->addChild($1);

```

```

    t1->addChild($3);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| '-' expr %prec UNARYMINUS
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::UNARYMINUS, nullptr);
    t1->addChild($2);
    $$ = t1;
}
| NOT expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::NOT, nullptr);
    t1->addChild($2);
    $$ = t1;
}
| primary_expr { $$ = $1; }
| lit { $$ = $1; }
;

//Storage of binary operator
binary_operator : '+' { $$ = '+'; }
| '-' { $$ = '-'; }
| '/' { $$ = '/'; }
| '*' { $$ = '*'; }
| '%' { $$ = '%'; }
| AND { $$ = yy::parser::token::AND; }
| OR { $$ = yy::parser::token::OR; }
| XOR { $$ = yy::parser::token::XOR; }
| EQ { $$ = yy::parser::token::EQ; }
| NEQ { $$ = yy::parser::token::NEQ; }
| GT { $$ = yy::parser::token::GT; }
| GEQ { $$ = yy::parser::token::GEQ; }
| LS { $$ = yy::parser::token::LS; }
| LEQ { $$ = yy::parser::token::LEQ; }
;

//Storage of assignment expression
assignment_expression : primary_expr '=' expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>('=', nullptr);
    t1->addChild($1);
    t1->addChild($3);
    $$ = t1;
}
| primary_expr '=' '{' table_list '}'
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>('=', nullptr);
    t1->addChild($1);

```

```

    t1->addChild($4);
    $$ = t1;
}
;

//Storage of identifying attributes
primary_expr : ID
{
    $$ = make_shared<stibbons::Tree>(yy::parser::token::ID,$1);
}
| primary_expr '.' ID
{
    auto t1 = make_shared<stibbons::Tree>(yy::parser::token::ATT_ID,$3)
    ;
    t1->addChild($1);
    $$ = t1;
}
| primary_expr '[' expr ']'
{
    auto t1 = make_shared<stibbons::Tree>(yy::parser::token::TAB_ID,
        nullptr);
    t1->addChild($1);
    t1->addChild($3);
    $$ = t1;
}
| primary_expr '[' ']'
{
    auto t1 = make_shared<stibbons::Tree>(yy::parser::token::TAB_ID,
        nullptr);
    t1->addChild($1);
    $$ = t1;
}
| ID '(' ')'
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::CALL,$1);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| ID '(' initializer_list ')'
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::CALL,$1);
    t1->setPosition({@1.begin.line,@1.begin.column});
    t1->appendChildren($3);
    $$ = t1;
}
;

//Storage for table object

```

```

table_list : expr_list { $$ = $1; }
| pair_list { $$ = $1; }
| { $$ = make_shared<stibbons::Tree>(yy::parser::token::TABLE, nullptr); };

expr_list : expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::token::TABLE, nullptr);
    t1->addChild($1);
    $$ = t1;
}
| expr_list ',' expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::token::TABLE, nullptr);
    t1->appendChildren($1);
    t1->addChild($3);
    $$ = t1;
}
;

pair_list : pair
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::token::TABLE, nullptr);
    t1->addChild($1);
    $$ = t1;
}
| pair_list ',' pair
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::token::TABLE, nullptr);
    t1->appendChildren($1);
    t1->addChild($3);
    $$ = t1;
}
;

pair : expr ':' expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::token::PAIR, nullptr);
    t1->addChild($1);
    t1->addChild($3);
    $$ = t1;
}
;

//Storage of call expression
initializer_list : expr

```

```

{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
    token::CALL, nullptr);
  t1->addChild($1);
  $$ = t1;
}
| initializer_list ',' expr
{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
    token::CALL, nullptr);
  t1->appendChildren($1);
  t1->addChild($3);
  $$ = t1;
}
;

//Storage for creation of new agent
creat_statement : nb_agt NEW AGT bloc
{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
    token::NEW, nullptr);
  t1->setPosition({@1.begin.line ,@1.begin.column});
  t1->addChild($1);
  t1->addChild($4);
  $$ = t1;
}
| nb_agt NEW ID '(' initializer_list ')'
{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
    token::NEW,$3);
  t1->setPosition({@1.begin.line ,@1.begin.column});
  t1->addChild($1);
  t1->addChild($5);
  $$ = t1;
}
| nb_agt NEW ID '(' ')'
{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
    token::NEW,$3);
  t1->addChild($1);
  t1->addChild(make_shared<stibbons::Tree>(yy::parser::token::NEW,
    nullptr));
  t1->setPosition({@1.begin.line ,@1.begin.column});
  $$ = t1;
}
| primary_expr '=' creat_statement
{
  stibbons::TreePtr t1 = make_shared<stibbons::Tree>('=', nullptr);
  t1->addChild($1);
  t1->addChild($3);
  $$ = t1;
}

```



```

}
;

nb_agt : NUMBER
{
    $$ = make_shared<stibbons::Tree>(yy::parser::token::NUMBER, $1);
}
| primary_expr
{
    $$ = $1;
}
|
{
    stibbons::ValuePtr nb = make_shared<stibbons::Number>(1.0);
    $$ = make_shared<stibbons::Tree>(yy::parser::token::NUMBER, nb);
}
;

// Storage of turtle instructions

instr_turtle : FD expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::FD, nullptr);
    t1->addChild($2);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
}
| LT expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::LT, nullptr);
    t1->addChild($2);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
}
| RT expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::RT, nullptr);
    t1->addChild($2);
    t1->setPosition({@1.begin.line, @1.begin.column});
    $$ = t1;
}
| PU
{
    $$ = make_shared<stibbons::Tree>(yy::parser::token::PU, nullptr);
}
| PD
{
    $$ = make_shared<stibbons::Tree>(yy::parser::token::PD, nullptr);
}

```

```

}
| SEND expr expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::SEND, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| SEND expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::SEND, nullptr);
    t1->addChild($2);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| RECV primary_expr primary_expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::RECV, nullptr);
    t1->addChild($2);
    t1->addChild($3);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| RECV primary_expr
{
    stibbons::TreePtr t1 = make_shared<stibbons::Tree>(yy::parser::
        token::RECV, nullptr);
    t1->addChild($2);
    t1->setPosition({@1.begin.line,@1.begin.column});
    $$ = t1;
}
| DIE
{
    $$ = make_shared<stibbons::Tree>(yy::parser::token::DIE, nullptr);
};

//Storage of literals
lit : NUMBER { $$ = make_shared<stibbons::Tree>(yy::parser::token::
    NUMBER,$1); }
| STRING { $$ = make_shared<stibbons::Tree>(yy::parser::token::STRING
    ,$1); }
| BOOLEAN { $$ = make_shared<stibbons::Tree>(yy::parser::token::
    BOOLEAN,$1); }
| COLOR { $$ = make_shared<stibbons::Tree>(yy::parser::token::COLOR,
    $1); }
| NIL { $$ = make_shared<stibbons::Tree>(yy::parser::token::NIL,$1);
    }
}

```

```
| TYPE { $$ = make_shared<stibbons::Tree>(yy::parser::token::TYPE, $1)
| ; };

%%

/*
 * Editor modelines  -  http://www.wireshark.org/tools/modelines.
 *    html
 *
 * Local variables:
 * mode: c++
 * c-basic-offset: 4
 * tab-width: 4
 * indent-tabs-mode: t
 * truncate-lines: 1
 * End:
 *
 * vim: set ft=cpp ts=4 sw=4 sts=4
 */
```

D.3 CppUnit

```
/*
 * This file is part of Stibbons.
 *
 * Stibbons is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as
 * published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Stibbons is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License
 * along with Stibbons. If not, see <http://www.gnu.org/licenses/>.
 */

#include "../model/agent.h"
#include "../model/zone.h"
#include "../model/turtle.h"
#include "../model/number.h"
#include "../model/string.h"
#include "../model/color.h"

#include <cppunit/TestCase.h>
#include <cppunit/extensions/HelperMacros.h>
using namespace stibbons;
using namespace std;
```

```

using namespace CppUnit;

class TestAgent : public TestCase {
    CPPUNIT_TEST_SUITE( TestAgent );
    CPPUNIT_TEST( getValuesT );
    CPPUNIT_TEST( getValuesZ );
    CPPUNIT_TEST( changeValueT );
    CPPUNIT_TEST( getValuesW );
    CPPUNIT_TEST_SUITE_END();

public :
    TurtlePtr t;
    ZonePtr z;
    WorldPtr w;

    void setUp() {
        auto worldSize = Size(2);
        worldSize.setValue(0, 10);
        worldSize.setValue(1, 10);
        auto zoneSize = Size(2);
        zoneSize.setValue(0, 10);
        zoneSize.setValue(1, 10);
        auto warp = vector<BorderType>();
        warp.push_back( BorderType::NONE );
        warp.push_back( BorderType::NONE );
        w=World::construct( worldSize , zoneSize , warp );
        w->setProperty( "tortue", make_shared<String>("bleu") );
        w->setProperty( "color", make_shared<Color>());

        z = Zone::construct( w );
        z->setProperty( "couleur", make_shared<String>("chat") );

        t=Turtle::construct( nullptr , w, 0 );
        t->setProperty( "chasse", make_shared<Number>(6.0) );
        t->setProperty( "couleur", make_shared<String>("chat") );
    }

    void getValuesT() {
        cout << "TestAgent::getValuesT" << endl;
        auto chasse = t->getProperty( "chasse" );
        CPPUNIT_ASSERT (Type::NUMBER == chasse->getType());
        auto chasse_reel = dynamic_pointer_cast<Number>( chasse );
        CPPUNIT_ASSERT_EQUAL (6.0, chasse_reel->getValue());
    }

    void changeValueT() {
        cout << "TestAgent::changeValueT" << endl;
        t->setProperty( "chasse", make_shared<Number>(7.7) );

        auto search = t->getProperty( "chasse" );
        CPPUNIT_ASSERT (

```

```

        search->getType() == Type::NUMBER &&
        dynamic_pointer_cast<Number>(search)->getValue() == 7.7
    );
}

void getValuesZ() {
    cout << "TestAgent::getValuesZ" << endl;
    auto couleur = z->getProperty("couleur");
    CPPUNIT_ASSERT (Type::STRING == couleur->getType());
    auto couleur_reel = dynamic_pointer_cast<String> (couleur);
    CPPUNIT_ASSERT ("chat" == couleur_reel->getValue());
}

void getValuesW() {
    cout << "TestAgent::getValuesW" << endl;
    auto couleur = w->getProperty("color");
    CPPUNIT_ASSERT (Type::COLOR == couleur->getType());
    auto tortue = w->getProperty("tortue");
    auto tt = dynamic_pointer_cast<String> (tortue);
    CPPUNIT_ASSERT ("bleu" == tt->getValue());
}

};

/* enregistrement du nom des test dans le registre */
CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(TestAgent, "TestAgent");

/*
 * Editor modelines  -  http://www.wireshark.org/tools/modelines.html
 *
 * Local variables:
 * c-basic-offset: 4
 * tab-width: 4
 * indent-tabs-mode: t
 * truncate-lines: 1
 * End:
 *
 * vim: set ft=cpp ts=4 sw=4 sts=4
 */

```

D.4 Json Spirit

```

{
    "time" : "Mon Jun  1 17:53:28 2015\n",
    "World" : {
        "WorldSize" : [
            3.0000000000000000,
            3.0000000000000000
        ],
        "ZoneSize" : [
            10.000000000000000,
            10.000000000000000
        ]
    }
}

```

```

],
"properties" : {
},
"Turtles" : [
  {
    "1" : {
      "color" : "#000000",
      "angle" : 0.0000000000000000,
      "position" : [
        0.0000000000000000,
        0.0000000000000000
      ],
      "properties" : {
      },
      "parent" : "world"
    }
  },
  {
    "0" : {
      "Breed" : "lila",
      "color" : "#000000",
      "angle" : 0.0000000000000000,
      "position" : [
        0.0000000000000000,
        0.0000000000000000
      ],
      "properties" : {
        "find" : true,
        "name" : "Kelly",
        "a" : 4.0000000000000000
      },
      "parent" : "world"
    }
  }
],
"Zones" : [
  {
    "0" : {
      "color" : "#ffffff",
      "properties" : {
      }
    }
  },
  {
    "1" : {
      "color" : "#ffffff",
      "properties" : {
      }
    }
  }
],
{

```

```

    "2" : {
      "color" : "#ffffff",
      "properties" : {
      }
    },
    {
      "3" : {
        "color" : "#ffffff",
        "properties" : {
        }
      },
      {
        "4" : {
          "color" : "#ffffff",
          "properties" : {
          }
        },
        {
          "5" : {
            "color" : "#ffffff",
            "properties" : {
            }
          },
          {
            "6" : {
              "color" : "#ffffff",
              "properties" : {
              }
            },
            {
              "7" : {
                "color" : "#ffffff",
                "properties" : {
                }
              },
              {
                "8" : {
                  "color" : "#ffffff",
                  "properties" : {
                  }
                }
              }
            }
          }
        }
      ]
    }
  }
}

```

Bibliographie

- [bis, 2015] (2015). GNU Bison - The Yacc-compatible Parser Generator - GNU Project - Free Software Foundation. <http://www.gnu.org/software/bison/>. 15
- [Aho et al., 2007] Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007). *Compilateurs : Principes, techniques et outils*. Pearson Education. 25
- [Ferber, 1995] Ferber, J. (1995). *Les Systèmes Multi Agents : vers une intelligence collective*. InterEditions. 5
- [Navarro, 2003] Navarro, J. (2003). Unit testing with CppUnit. <http://www.codeproject.com/Articles/5660/Unit-testing-with-CPPUnit>. 13
- [Paxson, 2014] Paxson, V. (2014). flex : The Fast Lexical Analyzer. <http://flex.sourceforge.net/>. 14
- [Pea, 1987] Pea, R. D. (1987). Logo programming and problem solving. *HAL : archives-ouvertes*. 7
- [Resnick et al., 2008] Resnick, M., Klopfer, E., et al. (2008). StarLogo on the web. <http://education.mit.edu/starlogo/>. 8
- [Stallman, 1988] Stallman, R. (1988). GDB : The GNU Project Debugger. <http://www.gnu.org/software/gdb/>. 14
- [Wilensky, 1999] Wilensky, U. (1999). NetLogo. <http://ccl.northwestern.edu/netlogo/>. 8
- [Wilkinson, 2014] Wilkinson, J. W. (2014). JSON Spirit : A C++ JSON Parser/Generator Implemented with Boost Spirit. <http://www.codeproject.com/Articles/20027/JSON-Spirit-A-C-JSON-Parser-Generator-Implemented>. 19