

ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS
Département Informatique
64 avenue Jean Portalis
37200 Tours, France
Tél. +33 (0)2 47 36 14 14
polytech.univ-tours.fr

Projet Recherche & Développement 2018-2019

Compilateur pour la robotique

Tuteur académique
Nicolas MONMARCHÉ

Étudiant
Maxence ROBIN (DI5)

4 avril 2019



Liste des intervenants

Nom	Email	Qualité
Maxence ROBIN	maxence.robin@etu.univ-tours.fr	Étudiant DI5
Nicolas MONMARCHÉ	nicolas.monmarche@univ-tours.fr	Tuteur académique, Département Informatique



Avertissement

Ce document a été rédigé par Maxence Robin susnommé l'auteur.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Nicolas Monmarche susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Maxence Robin, *Compilateur pour la robotique*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{  
  author={Robin, Maxence},  
  title={Compilateur pour la robotique},  
  type={Projet Recherche \& Développement},  
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},  
  address={Tours, France},  
  year={2018-2019}  
}
```

Document mis en forme sous L^AT_EX

grâce à la classe de document `polytech.cls` V2.2.5 (2017/05/02)



Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	v
1 Introduction	1
1 Acteurs, enjeux et contexte	1
2 Présentation et objectif.....	1
3 Hypothèses	2
4 Bases méthodologiques	2
2 Description générale	3
1 Environnement du projet	3
2 Caractéristiques des utilisateurs	3
3 Fonctionnalités du système	3
4 Structure générale du système	6

3	État de l'art	7
1	Blockly	7
1.1	Description et principe de base	7
1.2	Fonctionnement	7
1.3	Création de blocs personnalisés	8
1.4	Intégration	9
2	Scratch	9
3	Qt Creator	10
4	Visual Studio	11
5	Compilation	11
5.1	Flex/Bison	11
5.2	ANTLR	12
4	Analyse et conception	14
1	Choix des outils	14
1.1	Édition du programme avec Scratch et Blockly	14
1.2	Éditeur de code	15
1.3	Analyse lexicale et syntaxique avec Bison/Flex et ANTLR	15
2	Modélisation logicielle	15
3	Déroulement de la transformation.....	17
5	Mise en oeuvre	19
1	Compilateur.....	19
1.1	Installation d'ANTLR	19
1.2	Rédaction de la grammaire.....	20
1.3	Écriture du compilateur en C++	21
2	Interface	22
2.1	Création de l'interface principale	22
2.2	Éditeurs et exécuteurs	23
2.3	Intégration de Blockly	25

3	Transmission par USB	27
3.1	Présentation d'Ampy	27
3.2	Gestion dans l'application	28
6	Bilan et conclusion	29
	Annexes	31
A	Planification	32
B	Description des interfaces externes du logiciel	36
1	Interfaces matériel/logiciel	36
2	Interfaces homme/machine	36
3	Interfaces logiciel/logiciel	36
C	Spécifications fonctionnelles	37
1	Fonction : Création d'un nouveau programme.....	37
2	Fonction : Ouverture d'un programme existant	37
3	Fonction : Sauvegarde d'un programme	38
4	Fonction : Intégration de l'interface Blockly au sein de l'application.....	38
5	Récupération du code dans le format du langage pivot	39
6	Génération de l'arbre syntaxique abstrait.....	39
7	Génération du programme en MicroPython.....	39
8	Transmission du programme vers le robot	40
D	Spécifications non fonctionnelles	41
1	Contraintes de développement et conception	41
2	Contraintes de fonctionnement et d'exploitation	41
2.1	Performances.....	41
2.2	Capacités.....	41
2.3	Contrôlabilité.....	41
2.4	Sécurité	42

2.5	Intégrité.....	42
E	Guide d'utilisation	43
1	Création d'un programme.....	44
2	Ouverture d'un programme existant	45
3	Édition d'un programme.....	45
F	Cahier de développeur	49
1	Description des classes.....	49
1.1	Reprise de projet.....	52
1.1.1	Création de l'éditeur.....	54
1.1.2	Création de l'exécuteur.....	55
1.2	Créer de nouveaux blocs.....	55
1.3	Parser un langage.....	56
1.4	Créer le nouveau type de projet	56
2	Cahier de tests	56
G	Glossaire	58
H	Index	59
	Webographie	60

Table des figures

2 Description générale

1	Diagramme de cas d'utilisation	4
2	Diagramme d'activité	5
3	Schéma du système.....	6

3 État de l'art

1	Programme "Hello World!" dans l'éditeur de Blockly.....	8
2	Description de bloc en JSON.....	8
3	Description de bloc en javascript	9
4	Résultat produit.....	9
5	"Blockly Developer Tools" utilisé pour créer des blocs personnalisés	9
6	Programme Tetris réalisé avec Scratch.....	10
7	Liste des modules de Qt	10
8	Phases de traitement avec Flex et Bison.....	11
9	Main en C++ pour la génération d'un arbre syntaxique abstrait avec ANTLR	13

4 Analyse et conception

1	Tableau de comparaison entre Blockly et Scratch.....	14
---	--	----

2	Architecture MVC avec Qt pour l'application autonome.....	16
3	Diagramme de classes	17
4	Illustration d'un programme avec Blockly pour les robots.....	17
5	Code pivot généré pour l'instruction "avancer pendant"	18
5 Mise en oeuvre		
1	Règle décrivant une boucle while	20
2	Représentation sous forme de diagramme de la règle décrivant une boucle while..	20
3	Fichier de configuration du plugin ANTLR pour VS code.....	20
4	Méthode générant le code MicroPython associé à la règle de la boucle while.....	22
5	Aperçu de la fenêtre de dialogue de création de programme	23
6	Filtre des fichiers par extensions.....	24
7	Outil de création des blocs	26
8	Fonction <i>generator</i> du bloc "move"	26
A Planification		
1	Diagramme de Gantt du semestre 9.....	32
2	Diagramme de Gantt du semestre 10	33
E Guide d'utilisation		
1	Menu principal de l'application.....	43
2	Fenêtre de dialogue de création d'un programme.....	44
3	Fenêtre de dialogue d'ouverture d'un programme.....	45
4	Liste des types de programmes disponibles à l'ouverture d'un programme	45
5	Fenêtre d'édition d'un programme	46
F Cahier de développeur		
1	Diagramme des classes final	49
2	Menu contextuel pour ajouter une bibliothèque	53

3	Déclaration des types de projets dans l'application	56
---	---	----

1

Introduction

1 Acteurs, enjeux et contexte

L'école Polytech Tours a commencé la construction d'un ensemble de robots qui sont destinés à être programmés par les élèves de PeiP pour les années à venir. Ces robots utilisent une carte programmable via le langage Python. L'école souhaiterait que les élèves puissent programmer les robots de façon très simplifiée, en utilisant des langages de programmation visuels.

Le sujet de ce projet de recherche et développement comporte deux parties principales. La première qui est le coeur du projet, porte sur la conception d'un générateur de compilateur à partir de fichiers de grammaire. La deuxième porte sur le projet global, qui utilise ce générateur de compilateur pour permettre de traduire des programmes conçus avec un langage de programmation visuel simple, tels que Blockly ou Scratch, vers des programmes dans des langages de programmation tels que MicroPython qui pourront être implantés dans les futurs robots de l'école.

2 Présentation et objectif

L'objectif principal est de permettre la conception simplifiée de programmes avec des langages visuels, qui pourront ensuite être traduits dans un langage cible, principalement le MicroPython qui est le langage supportés par les cartes choisies pour la conception des robots, mais également d'autres langages comme Arduino pour de futurs robots par exemple.

Une application autonome devra être mise en place pour prendre en charge de la conception des programmes via un éditeur visuel jusqu'à l'envoi des programmes après compilation sur les cartes des robots.

L'enjeu final est que de jeunes élèves, comme des élèves de primaire, puissent utiliser l'application autonome pour programmer les robots d'ici l'année prochaine.

3 Hypothèses

La génération du compilateur se fera en utilisant ANTLR. Cependant cette technologie est moins documentée que Bison/Flex, si la conception du compilateur avec ANTLR n'aboutit pas, Flex et Bison pourront être utilisés pour générer les analyseurs lexicaux et syntaxique pour le projet.

La transmission du code doit se faire via l'application autonome directement vers les robots. Cette opération nécessite de communiquer sur les ports USB en utilisant un protocole de ports série. Comme cette opération peut être compliquée à réaliser, le programme en MicroPython pourra être simplement affiché dans une fenêtre afin que l'utilisateur puisse le copier/coller dans un logiciel externe permettant la transmission du code vers les robots.

4 Bases méthodologiques

1. Outils utilisés
 - Système d'exploitation : Windows 10
 - Gestion de projet : Trello
 - Langage de programmation : C++
 - Développement : Qt Creator
 - Rédaction du rapport : \LaTeX avec Overleaf
2. Méthode de gestion de projet Le projet est divisé en deux parties, la recherche et le développement. La partie recherche se déroule durant le semestre 9, durant cette phase, les différentes étapes sont les suivantes :
 - Rédaction des éléments de description généraux du projet
 - Recherche des outils et solutions existantes
 - Sélection des meilleures solutions
 - Rédaction de l'état de l'art
 - Rédaction des cahiers des charges et des spécifications

La partie développement se déroulera suivant une méthode orientée Agile, divisée en sprints qui auront pour but de fournir à chaque étape un produit fini. Les livrables attendus pour chaque sprint seront précisés, et les spécifications fonctionnelles associées également.

2

Description générale

1 Environnement du projet

Il n'y a pas d'environnement existant pour ce projet à proprement parler. Blockly constitue une partie existante à intégrer au sein de la future application. Son rôle est d'assurer la phase de création du programme et de la transformer vers le langage pivot qui servira alors de base pour le compilateur.

Des outils de compilation tels que ANTLR pourront également être intégrés pour faciliter les phases d'analyse lexicale et syntaxique.

2 Caractéristiques des utilisateurs

Les utilisateurs du projet seront de jeunes élèves comme des élèves en école primaire. La majorité d'entre-eux ne seront pas familiarisés avec l'informatique et n'auront jamais programmé. Il est donc nécessaire de leur faciliter la tâche au maximum et de proposer un environnement simple et efficace.

Les caractéristiques des utilisateurs peuvent être résumées dans le tableau suivant :

3 Fonctionnalités du système

Les fonctionnalités attendues du système sont les suivantes :

- Possibilité de créer un programme sous forme visuelle avec des blocs
- Possibilité de gérer les programmes créés avec un système de projets
- Traduction du programme vers un langage pivot
- Compilation du code à partir du langage pivot vers le langage de programmation à destination du robot (MicroPython)

- Possibilité de transmettre le code généré vers le robot, ou de pouvoir le copier pour le transmettre via un autre environnement existant.

Le diagramme de cas d'utilisation est montré sur la figure 1

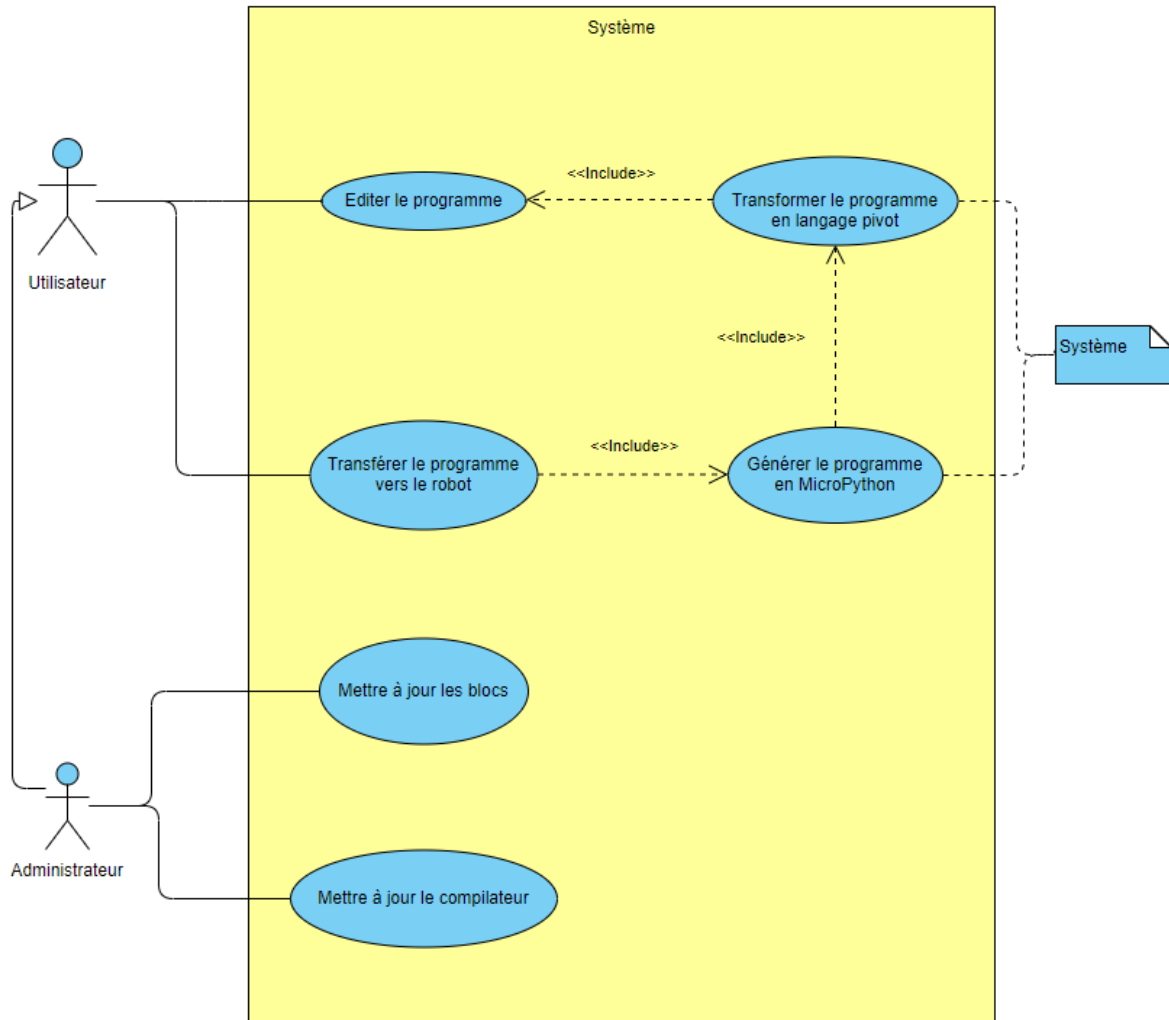


Figure 1 – Diagramme de cas d'utilisation

Deux utilisateurs humains sont présent, l'utilisateur classique et l'administrateur. L'utilisateur classique utilise l'application autonome pour programmer des robots. Il a à sa disposition la possibilité d'éditer un programme et de le transférer vers le robot cible. Le transfert vers le robot implique un traitement de la part du système, qui a la possibilité de transformer le programme Blockly en langage pivot avant de générer le programme final en MicroPython à partir de ce dernier.

L'administrateur quant à lui peut également mettre à jour le système en créant de nouveaux blocs et en adaptant le compilateur pour prendre en charge ces nouveaux blocs créés.

L'utilisation classique de l'application au cours d'une session peut être résumée par le diagramme d'activité présent sur la figure 2 :

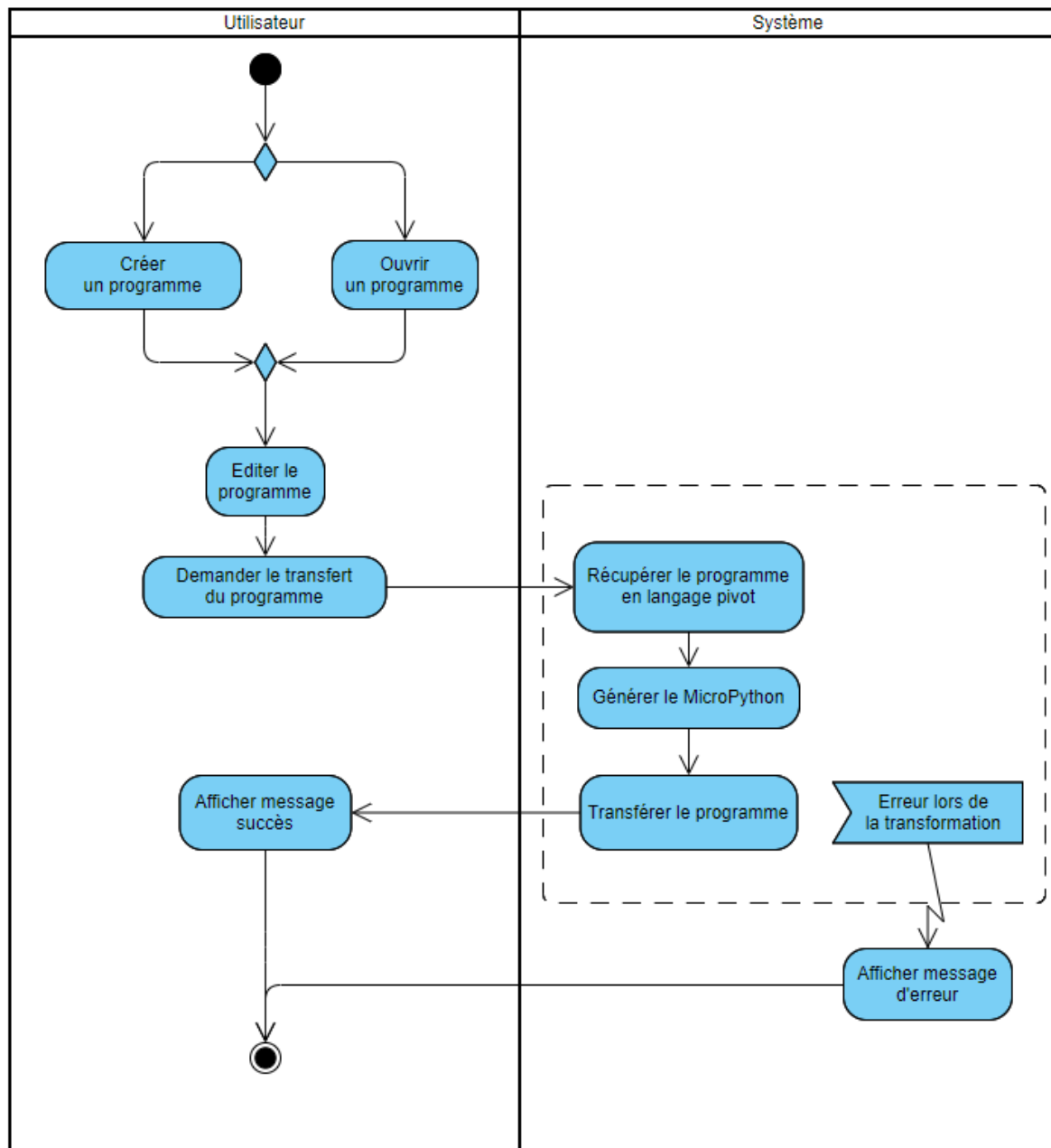


Figure 2 – Diagramme d'activité

Au lancement de l'application, l'utilisateur peut choisir de créer un nouveau programme ou d'en ouvrir un existant. Il peut alors modifier ce programme, et lorsque ce dernier est terminé, l'utilisateur peut alors essayer de transférer le programme vers le robot. La compilation du programme est alors lancée pour passer de Blockly au MicroPython. Si une erreur survient durant cette phase, la suite du traitement est interrompue et un message d'erreur indique à l'utilisateur ce qui pose problème. Dans le cas contraire si tout se déroule sans incident, un message indique que le transfert a été effectué avec succès et le robot est prêt à démarrer.

4 Structure générale du système

Le système global peut être représenté par la figure 3.

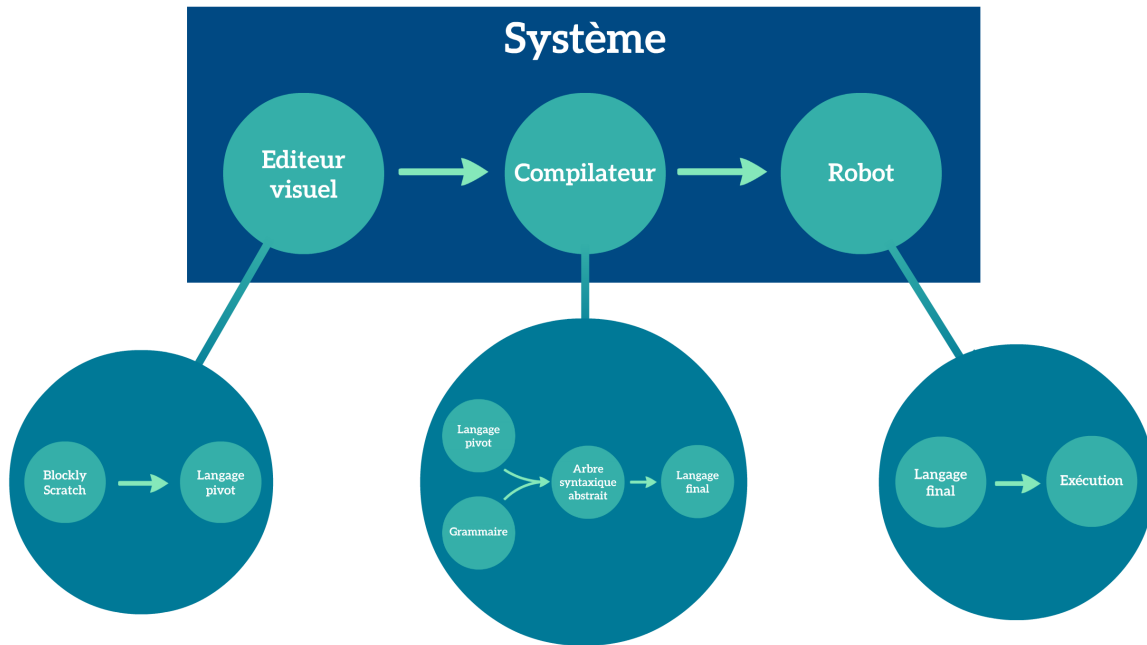


Figure 3 – Schéma du système

Le principe est le suivant : L'éditeur visuel permet de programmer avec un langage tel que Blockly ou Scratch. Le code est ensuite traduit vers un langage pivot qui représente de façon abstraite les actions qui devront être réalisées par le robot. L'objectif de ce langage est de ne pas être lié au robot, pour pouvoir s'adapter aux modifications à venir.

Une fois le code dans le langage pivot récupéré, il est envoyé au compilateur qui, à l'aide d'une grammaire, produit un arbre syntaxique abstrait. Cet arbre représente le code à traiter, on peut alors à partir de là traduire le code dans le langage de destination, c'est-à-dire Python ou Arduino. Le code est ensuite envoyé vers la carte du robot qui peut l'exécuter.

3

État de l'art

Les différents outils existants qui peuvent avoir une utilité pour le projet sont décrits ici.

1 Blockly

[[WWW1](#)]

1.1 Description et principe de base

Blockly est une bibliothèque réalisée par Google et écrite en Javascript, qui permet de réaliser des programmes visuellement avec un système de blocs qui représentent des instructions et qui peuvent s'emboîter les uns dans les autres. L'avantage de ce système est qu'il permet d'éviter toute erreur syntaxique puisque deux blocs qui ne sont pas compatibles ne pourront pas être emboîtés. Néanmoins d'autres problèmes tels que les boucles infinies peuvent toujours apparaître.

1.2 Fonctionnement

Blockly a été conçu pour être modulable et pour permettre de créer ses propres blocs afin de réaliser des langages dédiés, pour la robotique par exemple comme c'est le cas pour ce projet ainsi que BlocklyDuino, ou pour apprendre à programmer à de jeunes enfants grâce à Scratch Blocks. Pour chaque bloc créé, un code source est associé, une fois l'assemblage des blocs terminé, l'ensemble peut ainsi être converti dans un langage cible. Pour PRD l'objectif était donc de créer à la fois un ensemble de blocs pour Blockly adapté à la programmation des robots utilisés par les PeiP, et également de concevoir un langage pivot neutre, dédié au robot, qui pourrait ensuite être compilé vers le langage final, c'est-à-dire en Python ou en Arduino.

L'interface comporte trois zones. La zone principale est la zone centrale, c'est là que les blocs sont assemblés pour former le programme. Sur la partie gauche figure une liste de catégories,

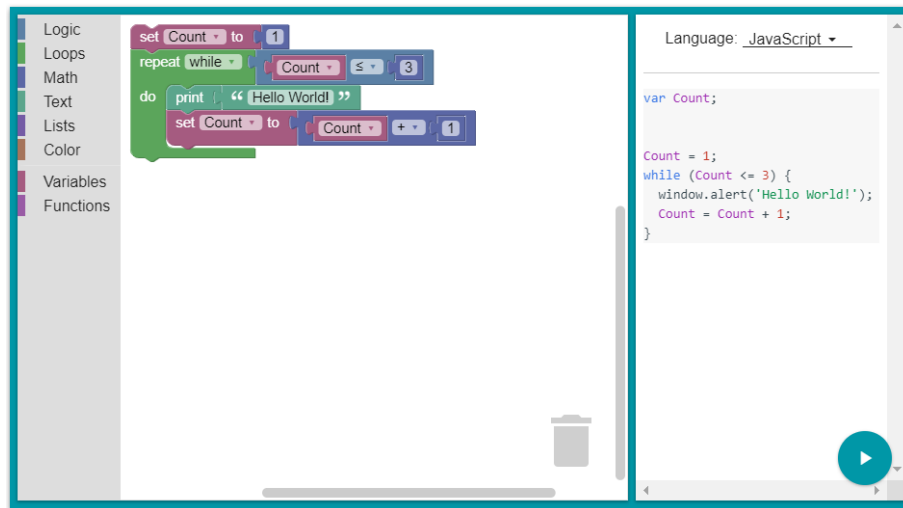


Figure 1 – Programme "Hello World!" dans l'éditeur de Blockly

qui renferment chacune plusieurs blocs. Enfin, la partie droite représente la traduction du programme vers le langage indiqué en haut.

Dans l'interface de Blockly originale (via un navigateur internet), il est possible d'exécuter le résultat directement en cliquant sur la flèche en bas à droite.

1.3 Création de blocs personnalisés

Les blocs personnalisés sont au coeur de Blockly, ils permettent de définir de nouveaux langages visuels qui pourront être traduits vers des langages de programmation classique ou tout autre format. Les blocs sont décrits avec le format JSON (figure 2) ou en javascript (figure 3), qui donne le bloc montré sur la figure 4.

```
{
  "type": "string_length",
  "message0": 'length of %1',
  "args0": [
    {
      "type": "input_value",
      "name": "VALUE",
      "check": "String"
    }
  ],
  "output": "Number",
  "colour": 160,
  "tooltip": "Returns number of letters in the provided text.",
  "helpUrl": "http://www.w3schools.com/jsref/jsref_length_string.asp"
}
```

Figure 2 – Description de bloc en JSON

Il est également possible de créer des blocs personnalisés grâce à un outil mis à disposition appelé "Blockly Developer Tools", montré sur la figure 5.

L'éditeur utilise lui-même l'éditeur visuel de Blockly pour définir les blocs personnalisés, le code JSON ou Javascript correspondant peut ensuite être généré pour être utilisé.

```
Blockly.Blocks['string_length'] = {
  init: function() {
    this.appendValueInput('VALUE')
      .setCheck('String')
      .appendField('length of');
    this.setOutput(true, 'Number');
    this.setColour(160);
    this.setTooltip('Returns number of letters in the provided text.');
```

```
    this.setHelpUrl('http://www.w3schools.com/jsref/jsref_length_string.asp');
  }
};
```

Figure 3 – Description de bloc en javascript



Figure 4 – Résultat produit

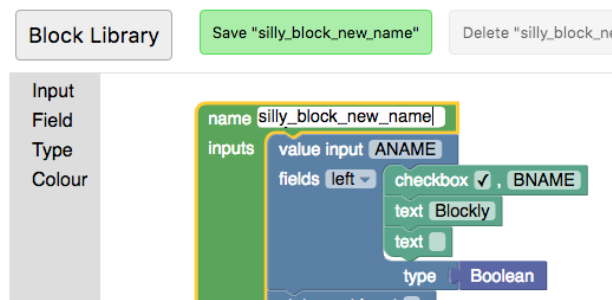


Figure 5 – "Blockly Developer Tools" utilisé pour créer des blocs personnalisés

1.4 Intégration

Blockly a été prévu pour être intégré sur trois plateformes différentes, à savoir pour le Web, Android et IOS. L'objectif de ce projet était de fournir une application autonome qui ne nécessite pas de connexion à internet pour fonctionner. La solution adoptée consiste donc à créer une interface graphique avec Qt, qui permet d'intégrer directement dans l'interface graphique un moteur web capable de faire tourner Blockly.

2 Scratch

Scratch est également un langage de programmation visuel par blocs, disponible avec son environnement de développement qui comporte des ressources en images, sons etc prêts à être utilisés. Scratch est principalement destiné à un public jeune, l'interface est donc pensée pour être utilisée par des enfants, comme on peut le voir sur la figure 6 qui représente le programme Tetris réalisé avec Scratch.

L'interface de Scratch est similaire à celle de Blockly, la zone à gauche contient les blocs regroupés par catégories, la zone centrale sert à l'assemblage des blocs pour former le programme.

La différence se situe dans la zone de droite, qui contient d'une part la zone d'exécution où se déroule le résultat du programme, et d'autre part une liste des ressources utilisées. Ces ressources peuvent être du son ou des images et peuvent être importées depuis la banque de

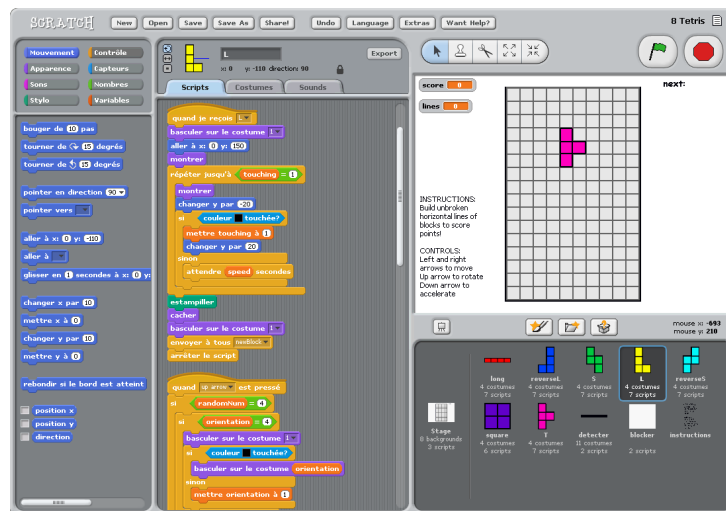


Figure 6 – Programme Tetris réalisé avec Scratch

données fournie ou depuis l'ordinateur, de plus il est possible de créer ses propres images directement via l'éditeur intégré à Scratch.

3 Qt Creator

[WWW3]

Qt Creator est un éditeur de code adapté à la bibliothèque Qt en C++. Les bibliothèques de Qt comportent de nombreux modules listés sur la figure 7.

Module	Description
Qt Core	Core non-graphical classes used by other modules.
Qt GUI	Base classes for graphical user interface (GUI) components. Includes OpenGL.
Qt Multimedia	Classes for audio, video, radio and camera functionality.
Qt Multimedia Widgets	Widget-based classes for implementing multimedia functionality.
Qt Network	Classes to make network programming easier and more portable.
Qt QML	Classes for QML and JavaScript languages.
Qt Quick	A declarative framework for building highly dynamic applications with custom user interfaces.
Qt Quick Controls 2	Provides lightweight QML types for creating performant user interfaces for desktop, embedded, and mobile devices. These types employ a simple styling architecture and are very efficient.
Qt Quick Dialogs	Types for creating and interacting with system dialogs from a Qt Quick application.
Qt Quick Layouts	Layouts are items that are used to arrange Qt Quick 2 based items in the user interface.
Qt Quick Test	A unit test framework for QML applications, where the test cases are written as JavaScript functions.
Qt SQL	Classes for database integration using SQL.
Qt Test	Classes for unit testing Qt applications and libraries.
Qt Widgets	Classes to extend Qt GUI with C++ widgets.

Figure 7 – *Liste des modules de Qt*

Les modules principaux, GUI et Widgets, sont utilisés pour la création des interfaces graphiques. D'autres modules permettent d'utiliser des fonctionnalités relatives au Web, d'interpréter des scripts ou de communiquer sur les ports série.

4 Visual Studio

Visual Studio est un éditeur de code réalisé par Microsoft qui peut être utilisé pour le développement en C++. Il est possible d'utiliser les bibliothèques de Qt avec Visual Studio en utilisant la version adaptée pour. Cependant, contrairement à Qt Creator, les bibliothèques de Qt ne sont pas directement intégrées et l'interface de l'éditeur n'est pas conçue spécifiquement pour les utiliser.

5 Compilation

Il existe déjà des outils permettant de s'occuper des phases de compilation, les principaux sont Flex/Bison et ANTLR. [WWW2]

5.1 Flex/Bison

Flex et Bison sont des outils libres permettant de créer des compilateurs. Flex est dédié à l'analyse lexicale, et Bison porte sur l'analyse syntaxique et sémantique.

La démarche pour utiliser ces outils est montrée sur la figure 8

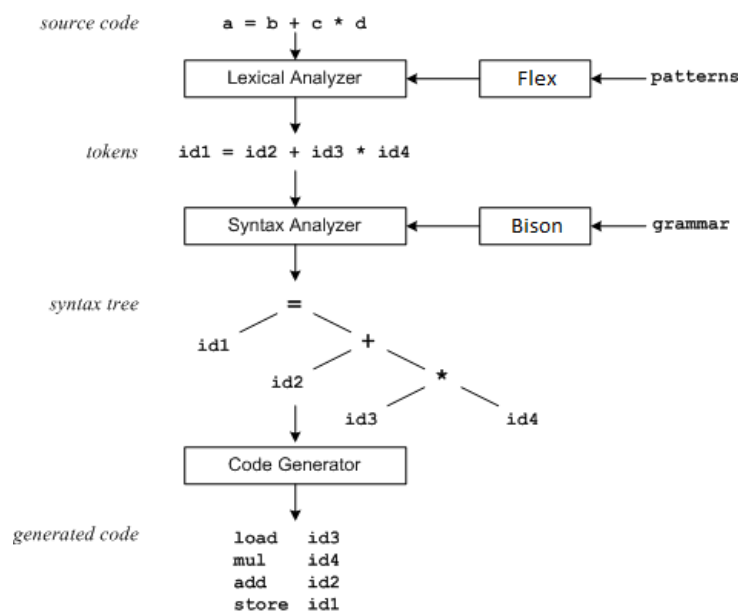


Figure 8 – Phases de traitement avec Flex et Bison

[WWW5]

Le principe est de créer une liste des différents tokens qui peuvent être rencontrés dans le fichier source, en les décrivant avec des expressions régulières (patterns). Avec un fichier de ce type, Flex est capable de produire un analyseur lexical qui pourra identifier les tokens trouvés à partir d'un fichier source. Le code source une fois analysé est transformé en une chaîne de tokens. Certains éléments sont remplacés par des symboles, c'est-à-dire une version abstraite. Par exemple tous les nombres comme "2", "45" etc seront remplacés par le symbole "nombre".

Le principe est similaire pour Bison, mais il faut fournir une grammaire, qui indique comment les différents tokens trouvés peuvent être regroupés pour former des entités. A partir de là un analyseur syntaxique peut être créé par Bison. La liste de tokens produite par l'analyseur lexical est envoyée à l'analyseur syntaxique qui à son tour la transformera en un arbre de syntaxe abstrait, c'est-à-dire une représentation du code source sous forme d'arbre. L'arbre contient les symboles qui ont été générés par l'analyseur lexical, pour retrouver les valeurs initiales qui étaient présentes dans le code source original, il faut passer par la "table des symboles", qui est simplement une table faisant le lien entre les symboles trouvés et les valeurs originales.

Une fois l'arbre produit il est possible de le parcourir pour réaliser diverses actions, la génération de code dans une langage de destination est l'une de ces actions possibles. Les actions réalisables lors de l'exploration de l'arbre sont décrites directement dans le fichier de description de la grammaire. Ainsi, lors de l'analyse de la chaîne de tokens par le parser, les actions sont réalisées au moment où les règles sont reconnues.

5.2 ANTLR

[[WWW4](#)]

ANTLR est un générateur de compilateur utilisant la notation augmentée de Backus-Naur. Contrairement à Lex et Bison, les actions à réaliser ne sont pas décrites directement dans le fichier de grammaire. A la place, les codes sources des lexers et des parseurs sont générés et peuvent être intégrés à un projet. Par exemple en C++, des classes correspondant aux analyseurs sont générées, il est alors possible de les inclure pour les utiliser afin de coder soi-même les actions à réaliser. Pour cela un patron de conception "Visiteur" est utilisé. Une fois l'arbre syntaxique abstrait généré, un visiteur le parcourt, et pour chaque noeud rencontré, du code peut être exécuté.

La procédure pour générer l'arbre syntaxique abstrait en C++ est telle que montrée sur la figure 9.

Le code source est d'abord converti au format ANTLR grâce à la classe `ANTLRInputStream`. Ce stream est ensuite passé en paramètre au lexer qui a été généré par ANTLR, il est alors possible de récupérer la liste des tokens générés en créant un objet de type `CommonTokenStream`. Cette liste de tokens est ensuite passée en paramètre au parseur généré, qu'on peut ensuite utiliser pour récupérer l'arbre syntaxique abstrait. Une classe `Visitor` est mise à disposition, mais il est nécessaire de créer sa propre classe héritant de la classe `Visitor` pour décrire comment l'arbre devra être exploré et quelles actions devront être réalisées pour chaque type de noeud rencontré. Dans l'exemple de la figure 9, la classe `ImageVisitor` est utilisée.

```

1  #include <iostream>
2
3  #include "antlr4-runtime/antlr4-runtime.h"
4  #include "antlr4-runtime/SceneLexer.h"
5  #include "antlr4-runtime/SceneParser.h"
6  #include "ImageVisitor.h"
7
8  using namespace std;
9  using namespace antlr4;
10
11 int main(int argc, const char* argv[]) {
12     std::ifstream stream;
13     stream.open("input.scene");
14
15     ANTLRInputStream input(stream);
16     SceneLexer lexer(&input);
17     CommonTokenStream tokens(&lexer);
18     SceneParser parser(&tokens);
19
20     SceneParser::FileContext* tree = parser.file();
21
22     ImageVisitor visitor;
23     Scene scene = visitor.visitFile(tree);
24     scene.draw();
25
26     return 0;
27 }

```

Figure 9 – Main en C++ pour la génération d'un arbre syntaxique abstrait avec ANTLR

4

Analyse et conception

1 Choix des outils

1.1 Édition du programme avec Scratch et Blockly

Les principes d'utilisation sont très proche de ceux de Blockly, néanmoins il existe des différences, qui sont résumées dans le tableau comparatif en figure 1.

	Scratch	Blockly
allows users to create custom blocks		X
can translate directly into other languages		X
makes syntax errors impossible	X	X
usable in your browser	X	X
downloadable on your PC	X	X
user community for code sharing	X	
great for making animations	X	
great for doing math	X	X
great for complex/abstract projects		X
put a coding environment on your webpage		X
useful for first-time coders	X	X

Figure 1 – Tableau de comparaison entre Blockly et Scratch

Les points majeurs qui jouent en faveur de Blockly sont :

- La création de blocs personnalisés, qui est nécessaire à la réalisation du langage pivot
- La traduction vers un langage dans un format choisi, pour les mêmes raisons

- La possibilité de gérer des projets plus complexes/abstraits, puisque ce projet rentre dans cette catégorie

La partie concernant le communauté n'est pas nécessaire dans notre cas puisqu'il existe de nombreuses ressources déjà mises à disposition. De plus il n'est pas possible d'intégrer l'éditeur de Scratch comme on peut le faire avec celui de Blockly, et il est préférable d'avoir à terme un environnement tout en un plutôt que de devoir passer par des éditeurs extérieurs.

Le choix final se porte donc vers Blockly qui est plus adapté à nos besoins.

1.2 Éditeur de code

Pour l'éditeur de code, Qt Creator semble être la solution adaptée pour les raisons suivantes :

- Les bibliothèques de Qt permettent de gérer les interfaces graphiques
- Nous avons déjà eu l'occasion de les utiliser, ce qui fait du temps d'apprentissage en moins
- Qt permet l'intégration de l'éditeur de Blockly grâce aux classes dédiées au Web
- La possibilité de communiquer sur les ports USB en utilisant un protocole de port série

1.3 Analyse lexicale et syntaxique avec Bison/Flex et ANTLR

Bison/Flex et ANTLR proposent des fonctionnalités similaires. Cependant ANTLR propose une syntaxe plus claire pour les fichiers de grammaire et puisque toute la partie concernant les actions à effectuer lorsque les règles sont reconnues sont séparées et sont mise en place dans le code C++ à la place. Cependant, cela implique de devoir les coder en C++ plus tard lorsque cela peut être fait plus rapidement avec Bison/Flex.

Il est nécessaire de pouvoir intégrer l'exécution de l'arbre syntaxique abstrait à l'ensemble de l'application, puisque celle-ci est une étape à part entière dans la chaîne de la création du programme avec Blockly vers l'envoi du programme transformé vers le robot.

Cela n'est pas possible avec Bison/Flex puisque le code généré est directement fait pour exécuter l'arbre, alors qu'ANTLR permet d'intégrer cette étape en tant que partie d'un projet.

C'est pour cette dernière raison qu'ANTLR semble être la solution la plus adaptée à ce projet.

2 Modélisation logicielle

Le but de ce projet étant de développer une application autonome avec une interface graphique, le pattern MVC sera pris pour base dans la conception mais sera réadapté pour plusieurs raisons :

- Les vues avec Qt peuvent être gérées directement via les classes d'interfaces graphiques sans avoir à coder ou presque
- Il n'y a pas de modèle à proprement parler ici puisque les seules données manipulées sont les programmes stockés sous forme de fichier sur l'ordinateur

Dans notre cas les vues seront donc gérées via le système de classes d'interfaces graphiques. Les contrôleurs hériteront directement de ces classes en y rajoutant simplement la logique pour former les "vue-contrôleurs". Bien qu'il n'y ai pas de base de données à manipuler, des classes de modèle seront utilisées pour représenter les objets "fichier" et faire l'interface entre le logiciel et l'ordinateur.

L'architecture est résumée sur le schéma de la figure 2

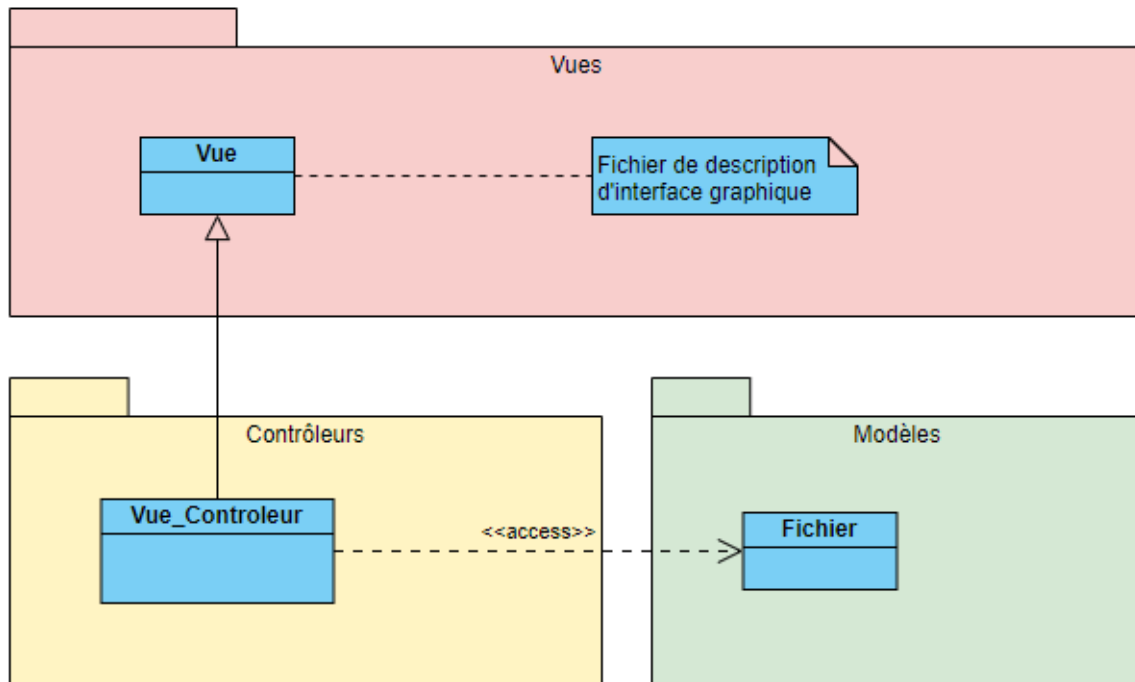


Figure 2 – Architecture MVC avec Qt pour l'application autonome

Les "vue-contrôleurs" contiennent le code des contrôleurs en héritant des vues, qui elles-mêmes utilisent le système de description d'interfaces graphique de Qt pour être gérées. Les fichiers sont représentés via des classes de modèle qui sont utilisées par les vue-contrôleurs.

Le diagramme de classes est montré sur la figure 3

On retrouve les deux vues qui représentent la fenêtre d'ouverture de programme et la fenêtre principales. Les deux vues sont généralisées pour former les deux vue-contrôleurs associés. La classe "ProgramFile" représente un programme Blockly, cette classe sert de modèle et fait l'interface avec le système de gestion de fichiers de l'OS, pour la création et la mise à jour des fichiers.

La classe MainViewController utilise la classe "Compiler" pour passer du code pivot récupéré directement depuis Blockly vers le MicroPython. Cette classe "Compiler" utilise les analyseurs lexical et syntaxique générés par ANTLR, ainsi que la classe "ASTVisitor" qui hérite de la classe "SceneBaseVisitor" fournie par ANTLR directement. Cette dernière classe va servir à parcourir l'arbre syntaxique abstrait et à écrire le résultat final en MicroPython.

Ce code MicroPython est retourné au MainViewController, qui le transmet ensuite au robot en utilisant la classe "USBTransmitter".

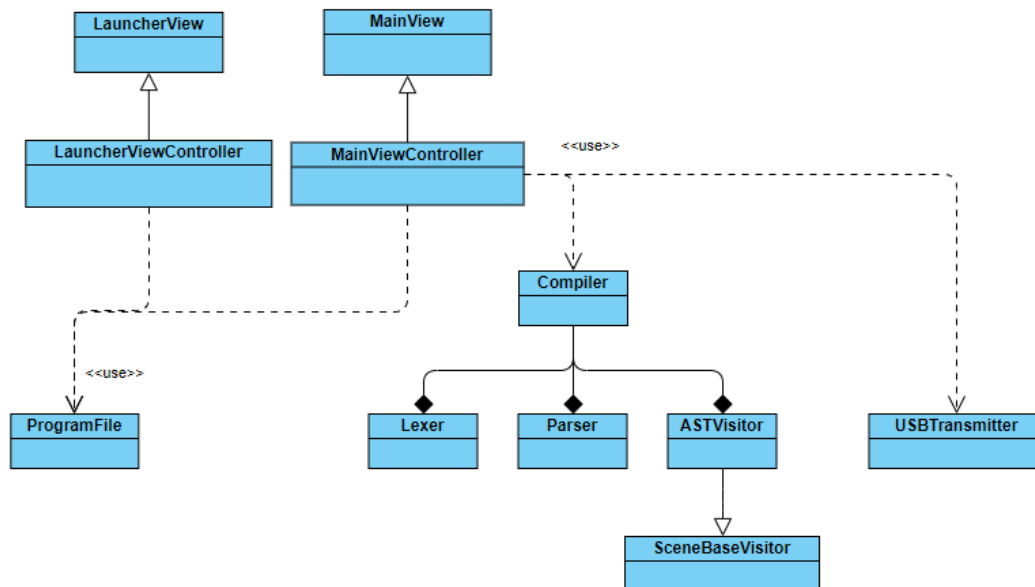


Figure 3 – Diagramme de classes

3 Déroulement de la transformation

La compilation via l'interface se déroulera comme montré sur les exemples qui suivent.

On considère l'instruction "avancer pendant" qui prend en paramètre le nombre de secondes pendant lesquelles le robot devra avancer en ligne droite.



Figure 4 – Illustration d'un programme avec Blockly pour les robots

La figure 4 montre d'une part à gauche le programme écrit directement dans l'interface de Blockly, et à droite la version traduite (code fictif) en MicroPython de ce programme. L'utilisateur peut ainsi voir à quoi ressemble le programme qui a été généré.

Entre temps le programme Blockly et le résultat en MicroPython, le code en langage pivot montré sur la figure 5 a été généré de façon transparente pour l'utilisateur.

```
1  avancer 10
2
3
4
```

Figure 5 – *Code pivot généré pour l'instruction "avancer pendant"*

La flèche en bas à droite permet la transmission du programme vers le robot.

5

Mise en oeuvre

1 Compilateur

La partie compilateur étant au coeur de l'application, il était important de pouvoir mettre en place rapidement celle-ci en place pour commencer les tests.

Il aurait été possible de créer une application totalement dépourvue d'interface et de tester les résultats de la traduction de langage pivot vers du MicroPython uniquement en passant des fichiers en paramètres, mais pour des raisons de facilité une interface basique a été mise en place contenant deux éditeurs de texte : Un pour écrire eu code en langage pivot et l'autre pour visualiser la traduction en MicroPython

De cette façon l'ordre d'implémentation des éléments était conservé, mais les modifications a effectuer pour la suite sur l'application au niveau de l'interface pour l'amener vers sa version finale étaient minimales.

Dans cette partie nous verrons comment le processus de traduction a été mis en place.

1.1 Installation d'ANTLR

La première chose à faire a été d'installer ANTLR et de configurer l'intégration de la grammaire avec l'application.

ANTLR est à la base une bibliothèque faite en Java, mais plusieurs portages dans de nombreux langages ont été faits. La version C++ a donc été utilisée. Pour utiliser ANTLR il faut télécharger la runtime pré-compilée et l'inclure dans les dépendances du projet. Le site officiel ne fournit qu'un runtime compilée en release, ce qui signifie qu'il se sera pas possible de compiler l'application qui l'utilise en debug.

Il est toutefois possilbe de télécharger les sources d'ANTLR directement et de les compiler soi-même pour générer une runtime en debug par exemple, ou bien avec le compilateur de son choix, ce qui n'est pas le cas par défaut puisque la runtime disponible sur le site a été compilée

avec MSVC 2015 64 bits ce qui signifie que ce même compilateur (ou un compatible comme MSVC 2017 64 bits) doit être utilisé pour compiler l'application également.

1.2 Rédaction de la grammaire

Il n'existe pas d'éditeur de code dédié à ANTLR, cependant plusieurs éditeurs différents peuvent être utilisés pour rédiger la grammaire. La compilation de la grammaire peut se faire en lignes de commandes, les fichiers générés dépendent que l'utilisation que l'on veut avoir.

Pour ce projet Visual studio code a été utilisé, car il existe un plugin ANTLR disponible pour cet éditeur, qui permet d'avoir une coloration syntaxique, une détection des erreurs ainsi que de pouvoir afficher différents graphes pour les règles de grammaire écrites, comme montré sur les figures 1 et 2.

```
117
118   while_loop : WHILE condition=boolean_expression SEP NEWLINE statements END;
119
```

Figure 1 – Règle décrivant une boucle while



Figure 2 – Représentation sous forme de diagramme de la règle décrivant une boucle while

Une fonctionnalité également utile est que le processus de compilation de la grammaire peut être automatisé via un fichier de configuration, comme montré sur la figure 3.

```
1  {
2      "antlr4.generation": {
3          "mode": "external",
4          "outputDir": "antlr4-runtime",
5          "language": "Cpp",
6          "listeners": false,
7          "visitors": true
8      },
9      "window.zoomLevel": 0
10 }
11
```

Figure 3 – Fichier de configuration du plugin ANTLR pour VS code

Cette configuration a pour but de décrire le processus de compilation à effectuer à chaque fois que de nouvelles modifications seront sauvegardées sur le fichier.

Le rôle des paramètres sont :

- `mode` : *external* indique que le code devra être généré pour pouvoir être utilisé pour une utilisation extérieure, ce qui est notre cas puisque nous voulons coder le comportement des règles en C++ dans l'application
- `outputDir` : Désigne le dossier dans lequel les fichiers seront générés relativement à l'emplacement du fichier de la grammaire lui-même
- `language` : *Cpp* indique que les fichiers générés devront être des classes C++
- `listeners` : Il existe deux types de fichiers qui peuvent être générés, des *listeners* et des *visitors*, la différence réside principalement dans le type d'événement détectés. Les *listeners* possèdent des méthodes qui sont appelées à chaque entrée et chaque sortie d'une règle, tandis que les *visitors* possèdent simplement une méthode appelée à chaque fois qu'une règle est reconnue. Ici *false* indique que les *listeners* ne seront pas générés
- `visitors` : *true* indique les *visitors* seront générés

1.3 Écriture du compilateur en C++

Il est possible dans l'écriture de la grammaire d'écrire des actions. Une action est un morceau de code associé à une règle, qui sera exécuté à chaque fois que cette règle sera reconnue. Il est ainsi possible faire, en partie, le travail de l'application directement dans ces actions.

Pour des raisons de clarté il a été décidé ne pas utiliser les actions pour avoir une grammaire contenant exclusivement les règles et aucun détail sur l'implémentation qui en sera faite. De cette façon la grammaire est plus lisible et sert uniquement à décrire le langage à reconnaître, tous les comportements associés à ces règles sont écrits dans le code C++ de l'application.

Au moment de la compilation de la grammaire, ANTLR génère des classes en C++ qui sont des visiteurs, c'est-à-dire que ces classes possèdent une méthode associée à chaque règle de la grammaire, qui sont appelées à chaque fois que la règle correspondante est reconnue. Les noms de ces méthodes sont générés de la façon suivante :

"visit" + <nom de la règle associée>

Dans le code de l'application, il est ensuite seulement nécessaire d'intégrer ces classes générées, puis de créer sa propre classe qui en hérite et de surcharger ces méthodes pour écrire le contenu, puisque par défaut ces méthodes ne font rien.

Dans le cas de notre application, le but est de traduire du code en langage pivot vers du MicroPython. Les règles de la grammaire sont déjà écrites pour reconnaître le langage pivot, il suffit donc d'écrire pour chaque règle le code qui va permettre de générer le code en MicroPython associé. Un exemple est montré sur la figure 4.

La variable *result* contient le code MicroPython représentant la boucle while. On commence par lui ajouter le mot clé "while" suivi de la représentation textuelle de l'expression booléenne qui suit. Comme cette expression est reconnue par une autre règle, le travail de traduction de celle-ci est délégué à la méthode associée, à savoir *visitBoolean_expression*. Cette méthode prend en paramètre l'expression booléenne nommée "condition" dans la règle montrée sur la figure 1.

À noter que le résultat renvoyé par cette méthode est converti en chaîne de caractères via méthode *as<string>()*. Cela est dû au fait que les méthodes générées par ANTLR retournent des objets de type *Any*, qui peuvent contenir n'importe quelle type de valeur, il est alors nécessaire

```

339 Any MicroPythonCompiler::visitWhile_loop(PivotParser::While_loopContext* context)
340 {
341     string result = "while " + visitBoolean_expression(context->condition).as<string>() + " :\n";
342
343     incrementIndentation();
344     result += visitStatements(context->statements()).as<string>();
345     decrementIndentation();
346
347     return std::move(result);
348 }

```

Figure 4 – Méthode générant le code MicroPython associé à la règle de la boucle *while*

de les convertir dans le type désiré pour pouvoir les exploiter.

Le même procédé est appliqué pour traduire la suite d'instructions de la boucle.

Les méthodes de traduction s'appellent donc entre elles, et c'est cette propagation des appels qui permet de traduire l'ensemble du programme. Puisque celui-ci est représenté sous forme d'arbre, chaque appel sous-appel de méthode permet en fait d'explorer les fils de noeuds de l'arbre jusqu'à arriver aux feuilles qui représentent les éléments terminaux de la grammaire décrit dans le lexer, les règles du parser qui effectuent des sous-appels sont les éléments non-terminaux de la grammaire.

2 Interface

L'interface était également essentielle puisque c'est le squelette de l'application, sans quoi elle serait inutilisable puisque le public visé concerne de jeunes élèves.

L'objectif a donc été d'essayer de faire une interface le plus simple d'utilisation possible, en minimisant les fonctionnalités futiles. Le plus gros défi a été de permettre de concevoir l'application de telle sorte que des modifications futures pour ajouter de nouvelles façons d'éditer les programmes ou de les exploiter soient facilement implémentables.

2.1 Création de l'interface principale

Pour l'interface nous avons choisi de faire une fenêtre principale qui servirait de base à l'application entière. Toujours pour éviter de perdre l'utilisateur, l'application est faite de telle sorte que seules les boutons permettant d'effectuer des actions qui ont un sens au moment actuel sont accessibles. Par exemple, en lançant l'application, seuls les boutons pour ouvrir et créer un programmes sont disponibles. Si un programme est ouvert, ces boutons sont désactivés, et il faudra commencer par fermer le programme en cours pour pouvoir en ouvrir un autre.

L'ouverture d'un programme repose entièrement sur le gestionnaire de fichier de l'OS lui-même, toujours pour faciliter l'utilisation.

Pour la partie création d'un nouveau programme, les informations suivantes sont à préciser :

- Le type de projet à créer : Différents types de projet/programme peuvent être créés, une liste des différents types disponibles est affichée, il suffit de cliquer sur le type voulu
- Le nom du programme à créer : Les caractères qui ne peuvent pas être utilisés pour des noms de fichier ne sont pas pris en compte ce qui permet d'avoir obligatoirement un nom

valide au moment de la création. De cette façon, cela évite de devoir afficher une fenêtre de dialogue pour indiquer que le nom n'est pas valide et faire choisir un nouveau nom à l'utilisateur.

- L'emplacement du programme sur l'ordinateur : Cette option est laissée facultative, une valeur par défaut est définie. Toutefois l'utilisateur peut choisir un autre emplacement que celui prédéfini, et s'il souhaite le définir comme le nouvel emplacement par défaut une simple case à cocher est mise à disposition

Toutes les informations doivent être renseignées pour que le bouton de création soit activé. L'application entière repose sur un système de détrompeur pour éviter d'avoir à afficher des fenêtres de dialogue pour indiquer à l'utilisateur qu'une valeur est mal renseignée. Tout cela dans le but de rendre l'utilisateur de l'application plus simple et agréable à utiliser.

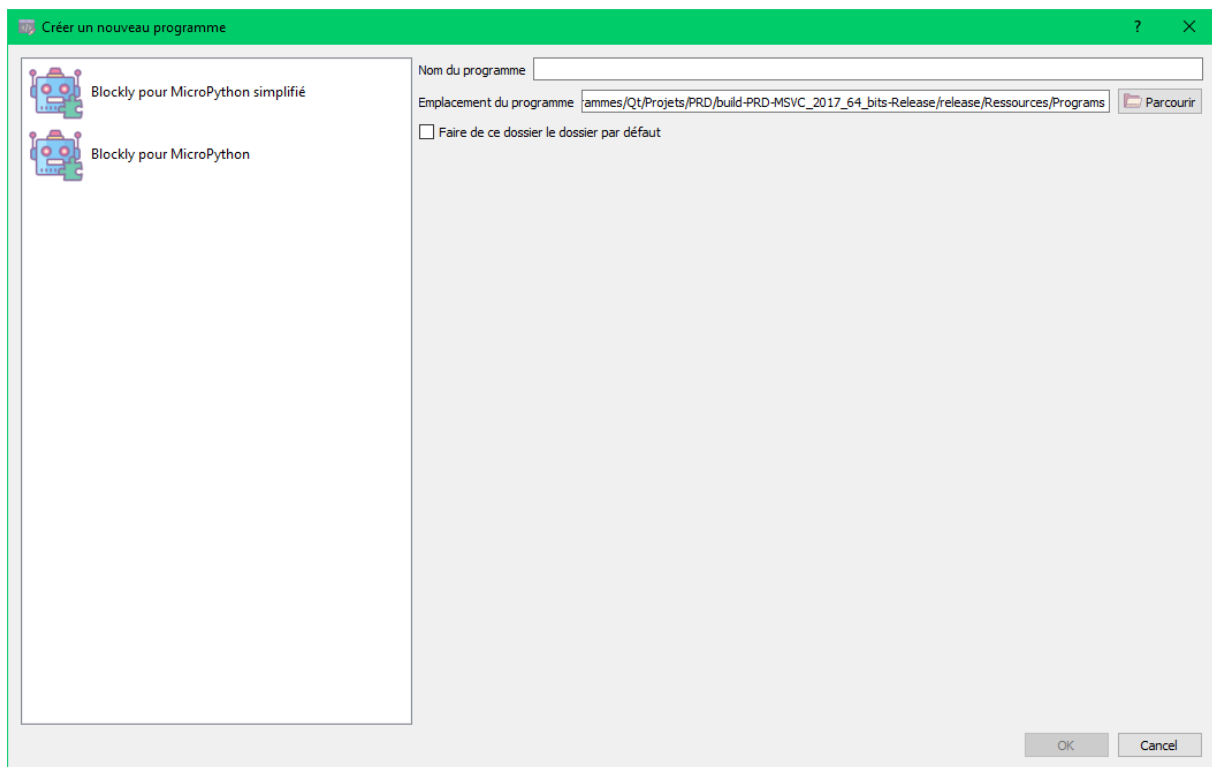


Figure 5 – Aperçu de la fenêtre de dialogue de création de programme

2.2 Éditeurs et exécuteurs

Une des contraintes du projet était de permettre dans le futur d'accepter d'autres types d'éditeurs que Blockly, et de pouvoir exploiter les programmes autrement qu'en les traduisant en MicroPython pour les envoyer sur les cartes Wipy, par exemple en les traduisant en C++ pour les envoyer vers une carte Arduino.

La partie la plus complexe de l'architecture de l'application en ce qui concerne l'interface a donc été de prévoir cela.

Nous avons donc choisi de créer deux classes abstraites, *AbstractEditor* et *AbstractExecutor*, qui allaient servir de base pour tous les futurs éditeurs et exécuteurs, qui auront pour rôle de permettre l'édition des programmes ou leur exploitation.

La fenêtre principale se content alors de gérer tout ce qui est générique, et dédie ces tâches en intégrant dans l'interface un éditeur et un exécuteur. Pour qu'ils puissent être intégrés, les deux classes *AbstractEditor* et *AbstractExecutor* héritent toutes les deux de la classe *QWidget* qui représente un conteneur vide dans l'interface.

Puisque les deux classes sont abstraites, elles possèdent certaines méthodes qui doivent être réimplémentées par le développeur.

Lorsque l'on crée un nouvel éditeur, il est nécessaire de réimplémenter une méthode appelée *getPivot()* qui retourne un code en langage pivot, de même lorsque l'on crée un nouvel exécuteur il faut réimplémenter la méthode *execute(pivot)* qui permet d'exploiter un code pivot passé en paramètre. Les éditeurs et exécuteurs sont donc indépendants et ne doivent pas avoir connaissance les uns des autres, ils doivent seulement avoir connaissance du langage pivot qui sert à les relier.

De cette façon il est possible d'interfacer différents éditeurs avec différents exécuteurs, et ce travail est réalisé par la fenêtre principale elle-même.

Le dernier problème à gérer concerne la façon dont on peut associer ces éditeurs et ces exécuteurs entre-eux.

Pour cela nous avons choisi de créer une classe appelée *ProjectType*, qui sert à décrire un type de projet et toutes les informations relatives à celui-ci.

C'est une classe template qui accepte deux types en "paramètres", pour former un couple éditeur/exécuteur. En plus de ce couple, cette classe enregistre l'extension de fichier associée, ainsi qu'une description du type de projet et une icône pour le décrire. Une liste d'objets de type *ProjectType* est ensuite créée, Cette liste est ensuite utilisée à divers endroits :

Au moment de l'ouverture d'un programme, les programmes sont rangés par type de projet en fonction de leur extension, comme montré sur la figure 6

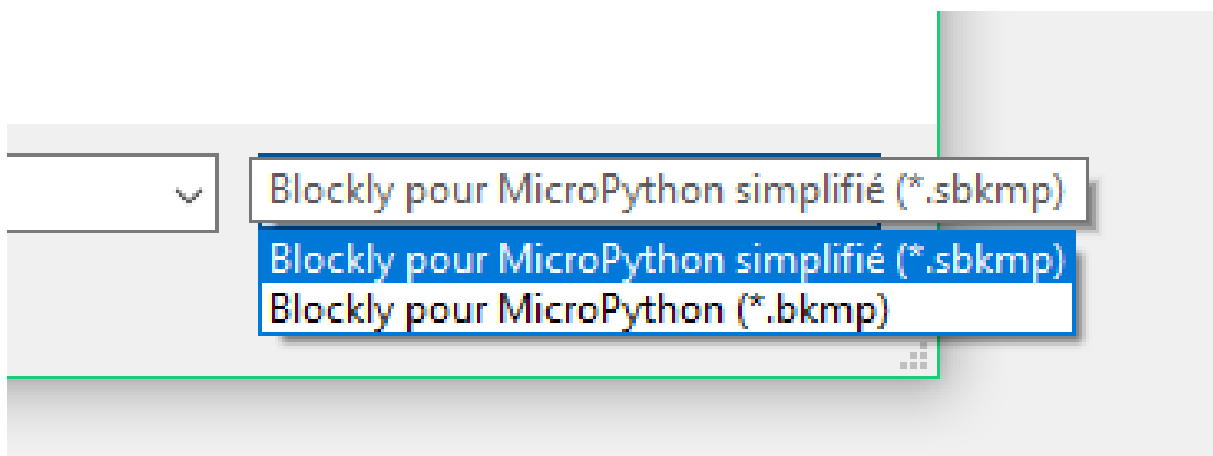


Figure 6 – Filtre des fichiers par extensions

Un filtre est créé pour chaque type présent dans la liste.

La liste permet également de créer la liste des types sélectionnables au moment de la création d'un programme.

De plus, lorsqu'un programme est ouvert, la fenêtre principale se charge de récupérer l'extension

du fichier ouvert, et peut ensuite consulter la liste pour retrouver l'objet de type *ProjectType* qui correspond à cette extension.

Elle peut alors avoir accès au couple éditeur/exécuteur. La classe *ProjectType* possède deux méthodes, *getNewEditor()* et *getNewExecutor()* qui renvoient respectivement un objet éditeur et exécuteur en fonction des types templates utilisés pour définir l'objet.

De cette façon, la fenêtre principale peut récupérer directement des objets du bon type à intégrer à l'interface à partir de l'extension d'un fichier.

La problématique est alors satisfaite, il est par exemple possible d'ajouter simplement un nouvel exécuteur pour envoyer les programmes vers des cartes Arduino simplement en définissant le comportement de l'exécuteur puis en créant un nouveau *ProjectType* faisant la paire entre un éditeur de type *BlocklyNeutralRobotEditor*, permettant d'éditer des programmes avec Blockly, et un exécuteur du type *ArduinoGenerator* qui est le nouveau type créé. Une fois ce *ProjectType* ajouté à la liste, toutes les fonctionnalités concernant la création d'un programme de ce type, la sauvegarde ainsi que l'ouverture sont automatiquement prises en compte.

2.3 Intégration de Blockly

L'intégration de Blockly était au coeur du projet, pour permettre de programmer simplement les robots.

Toutefois cette fonctionnalités n'a pas été directement implémentée comme une fonction de l'application en elle-même, mais comme un éditeur spécifique, c'est donc un cas particulier de l'utilisation de l'application.

L'éditeur met à profit la classe *QWebEngineView* de Qt qui est un moteur web permettant d'afficher des pages web à partir d'une URL mais également charger des fichiers en local. Cela est nécessaire puisque Blockly est une interface réalisée en langages web, HTML, CSS, Javascript.

Blockly est donc téléchargé et fait partie des ressources en local de l'application. Le moteur web se charge de charger l'interface dans l'application. L'avantage est que l'application ne nécessite pas d'avoir un accès à internet pour fonctionner.

Plusieurs contraintes ont dû être résolues vis-à-vis de l'intégration de Blockly :

Création des blocs personnalisés

La création des blocs a été réalisées avec un outil mis à disposition par Google, qui est montré sur la figure 7

La partie gauche sert à décrire le bloc. Sur la droite en haut, on peut avoir un aperçu du bloc en cours. En dessous, le code Javascript permettant de générer le bloc est affiché. Ce code est ensuite copié et collé dans un fichier javascript contenant les descriptions de tous les blocs personnalisés.

Tout en bas, un code javascript correspondant à la fonction *generator* est affiché. Ce code est également copié pour être ajouté dans un autre fichier javascript contenant la liste de toutes les fonctions *generator*.

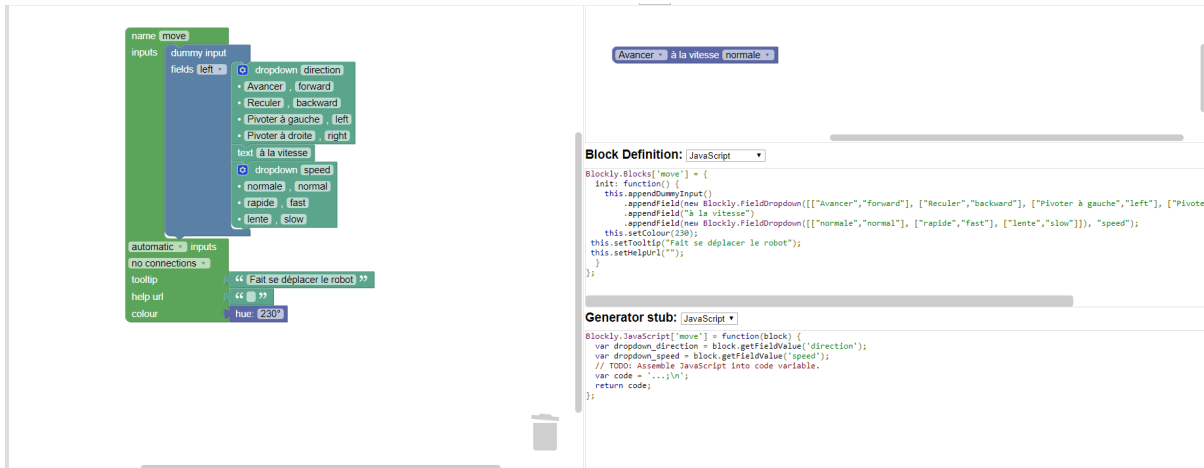


Figure 7 – Outil de création des blocs

Intégration des blocs personnalisés

Pour intégrer les blocs personnalisés, nous avons créé un fichier HTML représentant la vue principale. Cette vue intègre la zone de travail qui permet de placer et déplacer les blocs ainsi que sur la gauche une "boîte à outils". Cette boîte à outils est décrite dans le fichier HTML comme une suite de balise *category* qui contiennent des balises *block*. Chaque balise *category* permet de définir une des catégories principales comme "capteurs", "logique", "boucles" etc. Chaque balise *block* permet de faire référence à un bloc contenu dans le fichier javascript décrit précédemment. De cette façon nous pouvons intégrer les blocs créés auparavant à notre interface.

Traduction des blocs en langage pivot

La traduction des blocs en langage pivot s'effectue au sein des fonctions *generator*. Ces fonctions servent à récupérer les valeurs des entrées, et retournent une chaîne de caractères qui représente le bloc. Un exemple de traduction de bloc dans une fonction *generator* est montré sur la figure 8

```

4 // Move with a set of speed
5 Blockly.JavaScript['move'] = function(block) {
6   var dropdown_direction = block.getFieldValue('direction');
7   var dropdown_speed = block.getFieldValue('speed');
8
9   var code = dropdown_direction + " " + dropdown_speed + "\n";
10  return code;
11 };

```

Figure 8 – Fonction generator du bloc "move"

Les entrées sont converties sous forme de texte et concaténées, avant de renvoyer le tout.

Récupération du langage pivot généré

Blockly dispose de classes en Javascript qui sont utilisées dans l'interface du fichier HTML. La classe principale est la classe *Workspace*. Cette classe sert à manipuler la zone d'édition. Il existe une fonction *workspaceToCode(workspace)* qui prend en paramètre un *workspace* et retourne le code Blockly correspondant en faisant appel aux fonctions *generator*.

Il est possible avec *QWebEngineView* de lancer des appels de fonction en Javascript et d'en récupérer le résultat. La chaîne de caractères renvoyée par Blockly est ainsi récupérée dans le code C++ via l'appel de la fonction *workspaceToCode(workspace)*

3 Transmission par USB

Pour la transmission par USB, ici encore l'objectif était de pouvoir permettre de rajouter dans le futur la possibilité de gérer d'autres cartes qu'une Wipy. Pour cela nous avons créé une classe *AbstractSender* qui sert à communiquer avec des processus extérieurs. Mais cette classe n'implémente pas de comportement dédié pour communiquer avec un processus en particulier.

Nous avons ensuite créé une classe *WipySender* qui hérite de *AbstractSender*, spécifiquement pour envoyer les programmes sur la carte Wipy.

3.1 Présentation d'Ampy

Une première solution envisagée pour envoyer les programmes était d'utiliser la classe *QSerialPort* de Qt pour communiquer directement sur le port USB de la carte et ainsi lui transmettre les fichiers. Cependant cette solution requiert de connaître le protocole de communication utilisé par la carte, et nous n'avons pas réussi à le trouver.

Nous nous sommes donc tourné vers des bibliothèques qui permettent directement de communiquer avec une carte Wipy. Ampy est une bibliothèque en Python permettant de le faire. Il est possible de lancer des commandes en console pour effectuer différentes actions telles que :

- `put pcLocation/pcFile wipyLocation/wipyFile` : Envoie un fichier de l'ordinateur `pcFile` situé à l'emplacement `pcLocation` vers la carte wipy sous le nom `wipyFile` à l'emplacement `wipyLocation`
- `ls wipyLocation` : Liste tous les fichiers présents sur la carte à l'emplacement `wipyLocation`
- `reset` : Réinitialise la carte

Nous avons donc choisi d'adopter cette bibliothèque car elle répondait au besoin de l'application et était simple d'utilisation.

Pour transmettre les programmes, le code Blockly est donc traduit en MicroPython puis enregistré sur l'ordinateur dans un fichier appelé *main.py*. Ce fichier est ensuite envoyé sur la carte Wipy.

3.2 Gestion dans l'application

La classe *WipySender* est utilisée pour communiquer avec la bibliothèque Ampy en lançant des appels comme s'ils étaient fait en console. Pour cela la classe *WipySender* propose les mêmes fonctions qu'Ampy et sert donc d'interface.

Nous avons notamment implémenté les méthodes :

- `put(pcFile, wipyFile)` : Qui permet d'envoyer le fichier de l'ordinateur *pcFile* sur la carte Wipy sous le nom *wipyFile*. Le chemin est le fichier en lui même est inclu dans le nom du fichier
- `list(wipyLocation)` : Qui retourne la liste de tous les fichiers présents sur la carte Wipy à l'emplacement *wipyLocation*
- `reset()` : Qui réinitialise la carte

Cette classe est utilisée par l'exécuteur *MicroPythonGeneratorExecutor* pour envoyer le programme vers la carte une fois traduit.

6

Bilan et conclusion

Semestre 9

- Tâches terminées
 1. Description des fonctionnalités de base de l'application
 2. État de l'art
 3. Comparatif des technologies
- Tâches en cours
 1. Conception de l'architecture de l'application
 2. Définition des blocs personnalisés et du langage pivot
- Tâches restantes
 1. Création de l'interface
 2. Intégration de Blockly
 3. Création des blocs personnalisés
 4. Rédaction de la grammaire pour ANTLR
 5. Génération du MicroPython
 6. Transmission du code vers le robot
 7. Tests

Du retard a été pris au niveau de la conception du langage pivot et des blocs personnalisés pour Blockly. Une seule ou seulement très peu d'instructions seront donc utilisées dans un premier temps, avec l'objectif de parvenir jusqu'au bout du cycle de traduction de Blockly vers le MicroPython. Si l'objectif est atteint, de nouveaux blocs et instructions pourront alors être rajoutés facilement pour étoffer le langage.

Semestre 10

Le retard qui avait été pris lors du premier semestre a été rattrapé, et l'application est finalement fonctionnelle, de l'édition des programmes jusqu'à leur traduction et transmission vers le robot.

Ce projet a été l'occasion non seulement de mettre en oeuvre ce qui avait été appris durant les années précédentes mais également de découvrir de nouvelles technologies.

Globalement, ce projet est une réussite, bien que des améliorations peuvent encore être apportées pour perfectionner l'application.

Annexes

A

Planification

La première partie du projet est dédiée à l'état de l'art, ainsi que les description et les spécifications du projet.

Le projet est organisé avec une approche orientée Agile. Des sprints sont prévus pour la partie développement afin de fournir une application "complète" à chaque étape.

Les diagramme de Gantt prévus sont montrés sur les figures 1 et 2.

Semestre 9

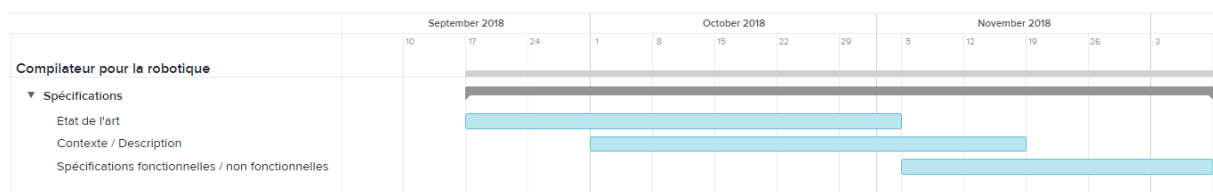


Figure 1 – *Diagramme de Gantt du semestre 9*

Résumé du diagramme de Gantt :

- Spécifications
 - 1. État de l'art
 - Date de début : 17 Septembre 2018
 - Date de fin : 5 Novembre 2018
 - Durée : 7 semaines
 - Description : Analyse des outils existants permettant pouvant être utiles au projet
 - 2. Contexte / Description
 - Date de début : 1er Octobre 2018
 - Date de fin : 19 Novembre 2018
 - Durée : 7 semaines

- Description : Description des enjeux et des objectifs, de l'architecture générale et de l'organisation du projet
- 3. Spécifications fonctionnelles / non fonctionnelles
 - Date de début : 5 Novembre 2018
 - Date de fin : 10 Décembre 2018
 - Durée : 5 semaines
 - Description : Description des spécifications fonctionnelles du projet, de l'organisation dans le temps et des contraintes relatives au développement

Semestre 10

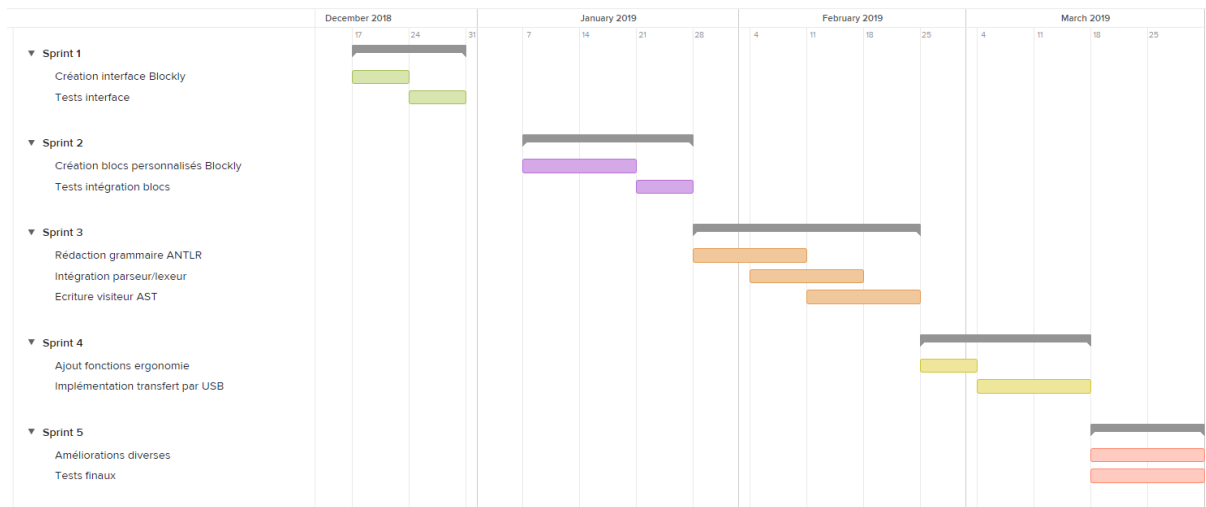


Figure 2 – Diagramme de Gantt du semestre 10

Résumé du diagramme de Gantt :

- Sprint 1

Le rôle de ce sprint est de fournir l'interface de base de l'application dans laquelle l'édition des programmes se fera

 1. Création interface Blockly
 - Date de début : 17 Décembre 2018
 - Date de fin : 24 Décembre 2018
 - Durée : 1 semaine
 - Description : Création de l'interface de base de l'application autonome, en intégrant Blockly
 2. Tests interface
 - Date de début : 24 Décembre 2018
 - Date de fin : 31 Décembre 2018
 - Durée : 1 semaine
 - Description : Tests de l'interface graphique
- Sprint 2

Ce sprint permet de fournir les blocs nécessaire à la programmation des premières instructions en rapport avec le robot cible via les blocs personnalisés correspondant

 1. Création blocs personnalisés Blockly
 - Date de début : 7 Janvier 2019
 - Date de fin : 21 Janvier 2019

- Durée : 2 semaines
- Description : Création des blocs personnalisés à intégrer à l'interface de Blockly, avec le code en langage pivot associé
- 2. Tests intégration blocs
 - Date de début : 21 Janvier 2019
 - Date de fin : 28 Janvier 2019
 - Durée : 1 semaines
 - Description : Tests de l'intégration des blocs à l'interface Blockly
- Sprint 3

Ce sprint permet la traduction du code en langage Blockly vers le MicroPython

 1. Rédaction grammaire ANTLR
 - Date de début : 28 Janvier 2019
 - Date de fin : 11 Février 2019
 - Durée : 2 semaines
 - Description : Écriture de la grammaire dans ANTLR pour générer les analyseurs lexical et syntaxique du langage pivot
 2. Intégration des analyseurs lexical et syntaxique
 - Date de début : 4 Février 2019
 - Date de fin : 18 Février 2019
 - Durée : 2 semaines
 - Description : Intégration des analyseurs lexical et syntaxique générés précédemment au projet, de sorte qu'il soit possible de générer une arbre syntaxique abstrait à partir d'un programme régigé avec Blockly ayant été traduit vers le langage pivot
 3. Écriture visiteur de l'arbre syntaxique abstrait
 - Date de début : 11 Février 2019
 - Date de fin : 25 Février 2019
 - Durée : 2 semaines
 - Description : Implémentation du code permettant à partir de l'arbre syntaxique abstrait généré précédemment d'écrire la traduction dans le langage cible (MicroPython dans un premier temps), en utilisant le patron de conception "visiteur"
- Sprint 4

Ce sprint a pour but de finaliser les fonctionnalités de l'interface après l'intégration des fonctions relatives aux blocs personnalisés introduites au sprint précédent

 1. Ajout des fonctions d'ergonomie
 - Date de début : 25 Février 2019
 - Date de fin : 4 Mars 2019
 - Durée : 1 semaine
 - Description : Ajout des fonctionnalités secondaires à l'application, comme les fonctions dédiées à la gestion des fichiers, à la sauvegarde et l'ouverture des programmes enregistrés sur l'ordinateur, pour finaliser les fonctionnalités de l'application autonome
 2. Implémentation du transfert par USB
 - Date de début : 4 Mars 2019
 - Date de fin : 18 Mars 2019
 - Durée : 2 semaines
 - Description : Implémentation du transfert du programme en MicroPython de l'application vers le robot via un câble USB
- Sprint 5

Ce sprint a pour but d'améliorer le système dans son ensemble, de finaliser l'application

 1. Améliorations diverses
 - Date de début : 18 Mars 2019

- Date de fin : 1 Avril 2019
 - Durée : 2 semaines
 - Description : Améliorations de diverses fonctionnalités du système ou ajout d'éventuelles fonctionnalités manquantes, et ajout de nouvelles instructions pour le robot
1. Tests finaux
 - Date de début : 18 Mars 2019
 - Date de fin : 1 Avril 2019
 - Durée : 2 semaines
 - Description : Tests finaux de l'application

B

Description des interfaces externes du logiciel

1 Interfaces matériel/logiciel

Les interfaces matériel/logiciel seront présentes entre l'application autonome et les robots. Les robots possèdent des carte ESP32, qui peuvent communiquer via des ports USB. Ce sont ces ports qui seront utilisés, pour envoyer les programmes en MicroPython. Le protocole utilisé est celui d'un port série émulé sur le port USB. La communication se fera en utilisant la classe QSerialPort de Qt, qui permet de d'envoyer et de recevoir des informations via les ports série.

2 Interfaces homme/machine

L'édition se fait via l'ouverture d'un document, soit en créant un nouveau programme soit en ouvrant un existant. L'accès à ces fonctionnalités se fait via le système de gestion de fichiers du système d'exploitation.

L'édition des programmes se fait directement via l'interface de Blockly qui est intégrée dans l'application autonome.

Les seuls éléments ajoutés à l'interface de Blockly sont le visualiseur de code et les menus d'accès à la création/ouverture de fichiers.

La communication sur l'échec ou le succès de la transmission du programme vers le robot se fera via une popup d'information.

3 Interfaces logiciel/logiciel

La gestion du stockage des données sera effectuée via des un système de fichiers pour sauvegarder les programmes rédigés par l'utilisateur.

C

Spécifications fonctionnelles

1 Fonction : Création d'un nouveau programme

Identification de la fonction

Cette fonction permet à l'utilisateur de créer un nouveau programme.

Priorité : **Secondaire**

Description de la fonction

Cette fonction prend en entrée le nom du programme et le chemin sur l'ordinateur où le programme devra être enregistré, et donne ensuite accès à l'interface d'édition de programmes, en enregistrant les informations rentrées par l'utilisateur pour les réutiliser au moment de la sauvegarde.

2 Fonction : Ouverture d'un programme existant

Identification de la fonction

Cette fonction permet à l'utilisateur d'ouvrir un programme existant sur le disque dur.

Priorité : **Secondaire**

Description de la fonction

Cette fonction affiche une fenêtre pour permettre à l'utilisateur de parcourir les fichiers présent sur l'ordinateur, en filtrant pour ne garder uniquement que les fichiers correspondant aux fichiers de type "programme blockly". Une fois sélectionné, le programme est chargé et restitué sur l'interface d'édition de programmes de Blockly et l'utilisateur peut de nouveau le modifier.

3 Fonction : Sauvegarde d'un programme

Identification de la fonction

Cette fonction permet à l'utilisateur d'enregistrer son programme en cours.

Priorité : **Secondaire**

Description de la fonction

Cette fonction permet à l'utilisateur d'enregistrer l'état actuel de son programme. Le fichier XML représentant le programme Blockly en cours d'édition est récupéré et enregistré sur l'ordinateur à l'emplacement choisi avec le nom choisi par l'utilisateur.

4 Fonction : Intégration de l'interface Blockly au sein de l'application

Identification de la fonction

Cette fonction permet de pouvoir éditer des programmes Blockly dans l'application.

Priorité : **Primordiale**

Description de la fonction

Cette fonction permet d'intégrer l'interface Web de Blockly au sein de l'application. L'intégration se fera en réalisant un fichier HTML représentant une page redimensionnable capable d'accueillir l'interface de Blockly. Cette page web sera ensuite intégrée dans l'interface grâce aux classes fournies par Qt.

5 Récupération du code dans le format du langage pivot

Identification de la fonction

Cette fonction permet de récupérer le programme écrit dans l'interface Blockly dans le format du langage pivot.

Priorité : **Primordiale**

Description de la fonction

Cette fonction prend en paramètre le programme écrit par l'utilisateur dans l'interface de Blockly et génère la version dans le langage pivot.

6 Génération de l'arbre syntaxique abstrait

Identification de la fonction

Cette fonction génère l'arbre syntaxique abstrait représentant le programme à partir du langage pivot.

Priorité : **Primordiale**

Description de la fonction

Cette fonction prend en entrée le programme au format langage pivot généré par la fonction "Récupération du code dans le format du langage pivot" (section 5), et génère l'arbre syntaxique abstrait correspondant à ce programme en utilisant les analyseurs lexical et syntaxique.

7 Génération du programme en MicroPython

Identification de la fonction

Cette fonction génère le programme en MicroPython à envoyer au robot.

Priorité : **Primordiale**

Description de la fonction

Cette fonction utilise l'arbre syntaxique abstrait généré par la fonction "Génération de l'arbre syntaxique abstrait" (section 6), et le parcourt pour créer le programme en MicroPython qui devra être envoyé au robot. Le programme généré pourra être affiché dans l'interface pour être consulté et copié par l'utilisateur.

8 Transmission du programme vers le robot

Identification de la fonction

Cette fonction permet d'envoyer le programme en MicroPython vers le robot via un port USB.

Priorité : **Secondaire**

Description de la fonction

Cette fonction prend en paramètre le programme en MicroPython généré par la fonction "Génération du programme en MicroPython" (section 7), et le transmet au robot via un câble USB.

D

Spécifications non fonctionnelles

1 Contraintes de développement et conception

- Langage : C++
- Bibliothèques : Qt, ANTLR, Blockly
- Environnement de développement : Qt Creator
- Matériel : Robots construits par Polytech, cartes ESP32

2 Contraintes de fonctionnement et d'exploitation

2.1 Performances

L'application autonome étant dédiée à la programmation des robots, et l'interface étant gérée directement par Blockly, il n'y a pas de contrainte de performance particulière. Seul le temps de traduction du programme de Blockly vers MicroPython doit se réaliser dans un temps raisonnable.

2.2 Capacités

Il n'y a pas de contrainte de capacité particulière.

2.3 Contrôlabilité

Pour la contrôlabilité un système de fichiers de log pourra être mis en place pour récupérer les différentes versions du programme depuis le code Blockly, la version en langage pivot et enfin en MicroPython pour détecter les éventuelles erreurs de traduction qui pourraient causer un dysfonctionnement.

2.4 Sécurité

Il n'y a qu'un seul type d'utilisateur et pas de système de sécurité particulier mis en place. Une fois sur l'application autonome, tout utilisateur peut ouvrir et modifier les fichiers qui sont présents sur l'ordinateur.

2.5 Intégrité

Un système de sauvegarde automatique à intervalle régulier pourra être mis en place pour permettre à l'utilisateur de récupérer le projet dans l'état auquel il était avant une éventuelle fermeture imprévue.

E

Guide d'utilisation

Au lancement de l'application, la fenêtre de la figure 1 s'ouvre :

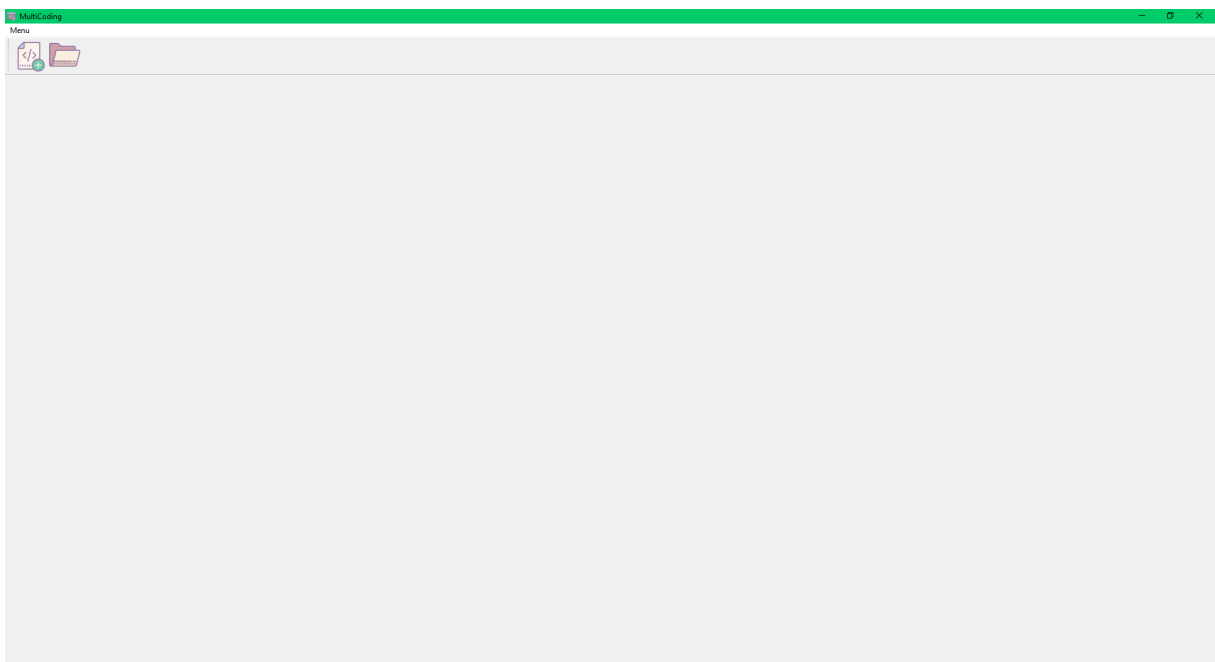


Figure 1 – *Menu principale de l'application*

C'est le menu principal. Elle est composée d'une barre de menu, d'une barre d'outils et d'une zone d'édition.

La barre de menu comporte un seul bouton "menu" qui permet seulement de quitter l'application.

La zone d'édition est vide puisqu'aucun programme n'est ouvert.

La barre d'outils est l'élément qui nous intéresse le plus ici. Elle comporte deux boutons, le premier permet de créer un nouveau programme, et le second permet d'ouvrir un programme existant, nous verrons en détails l'utilisation de ces deux boutons par la suite.

1 Création d'un programme

Lorsque l'on clique sur le bouton pour créer un programme, la fenêtre montrée en figure 2 s'affiche :

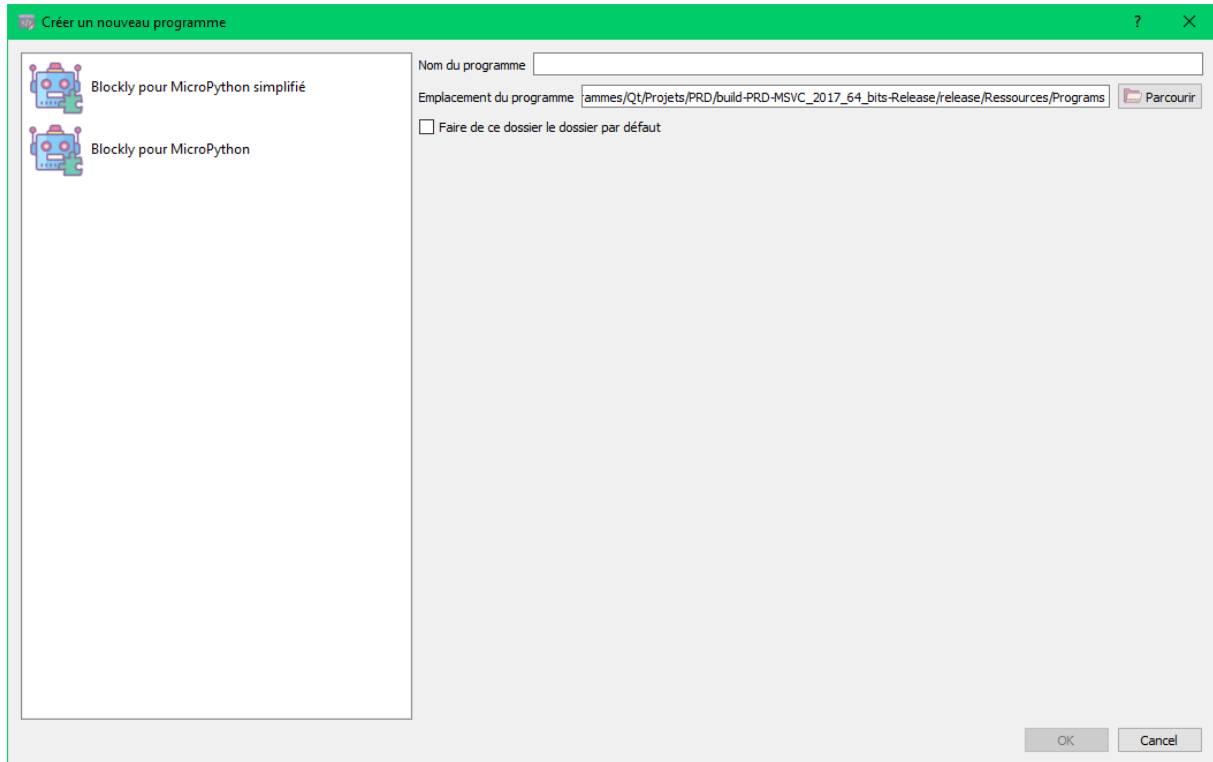


Figure 2 – Fenêtre de dialogue de création d'un programme

Plusieurs informations doivent être renseignées pour créer un nouveau programme. La première est le type de programme que l'on veut créer. Une liste des différents types de programmes disponibles est présente sur la gauche, ici "*Blockly pour MicroPython simplifié*" et "*Blockly pour MicroPython*". Une fois le type choisi, il est nécessaire de renseigner le nom du programme.

La dernière information à indiquer est le dossier dans lequel le programme sera créé. Il n'est pas nécessaire de préciser cette information, par défaut les programmes seront enregistrés dans le dossier "*Ressources/Programs*" qui se situe dans le dossier d'installation de l'application. Il est possible de choisir un nouveau dossier en cliquant sur "*Parcourir*". Si vous choisissez un nouveau dossier que celui par défaut, il est possible de cocher la case "*Faire de ce dossier le dossier par défaut*", une fois le programme créé le dossier choisi sera celui par défaut pour la création des prochains programmes.

Si un programme du même nom et du même type, se trouvant déjà dans le même dossier est présent au moment de la création d'un nouveau programme, le programme existant sera ouvert à la place.

2 Ouverture d'un programme existant

Si vous avez déjà créé un programme, vous pouvez l'ouvrir à partir du menu principal en cliquant sur le bouton avec une icône de dossier. Lorsque vous cliquez la fenêtre montrée sur la figure 3 apparaît :

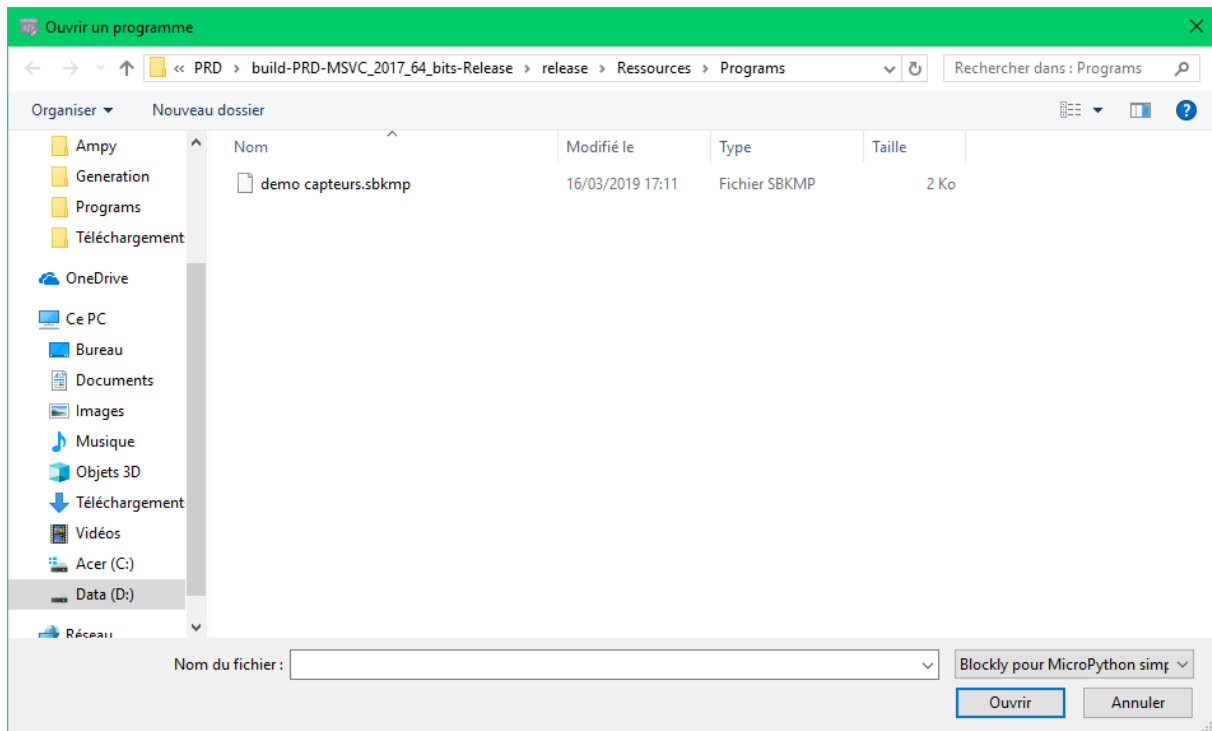


Figure 3 – Fenêtre de dialogue d'ouverture d'un programme

La fenêtre s'ouvre à l'emplacement choisi comme dossier par défaut pour la création des programmes (voir la partie sur la création des programmes). Il est important de noter que les programmes sont rangés par type, c'est-à-dire que seuls les programmes du même type sont affichés en même temps. Pour trouver le programme que vous recherchez il faut donc tout d'abord choisir le type, comme montré sur la figure 4.

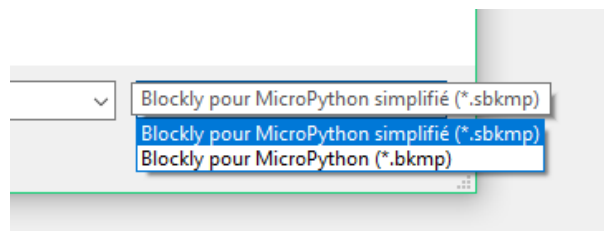


Figure 4 – Liste des types de programmes disponibles à l'ouverture d'un programme

3 Édition d'un programme

L'éditeur de programme dépend du type de programme créé. Ce guide présentera seulement les fonctionnalités disponibles pour l'éditeur Blockly pour le MicroPython.

La fenêtre d'édition des programmes est telle que montrée que la figure 5 :

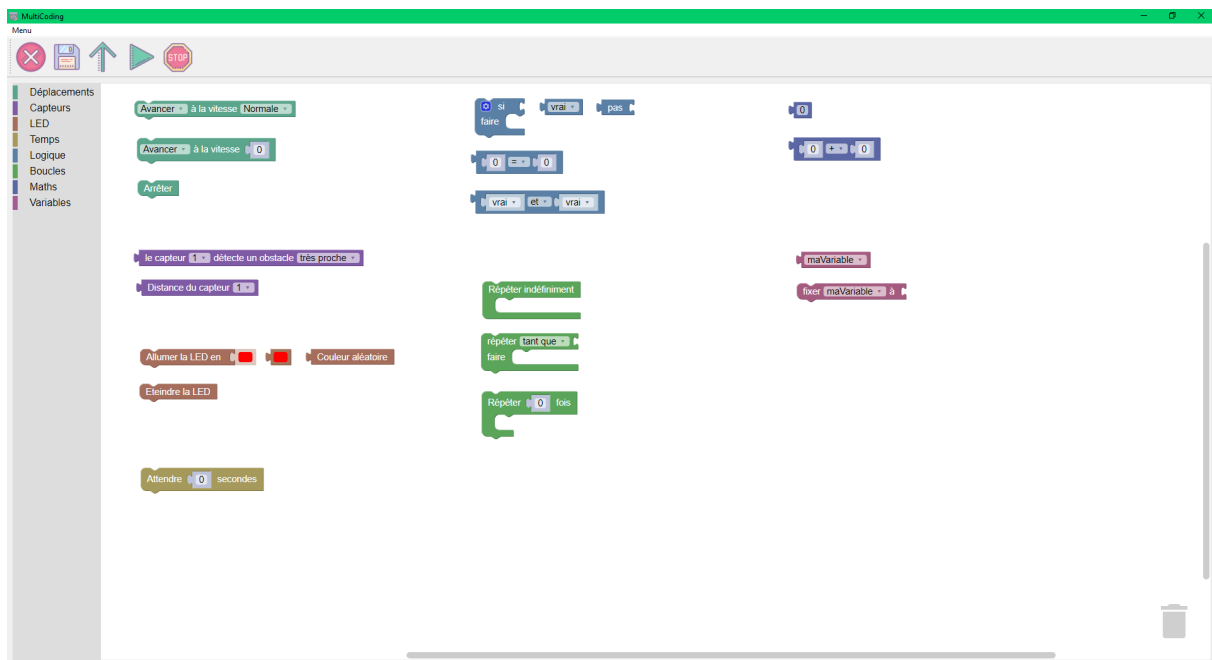


Figure 5 – Fenêtre d'édition d'un programme

Les icônes de la barre d'outils ne sont plus les mêmes. Le premier bouton avec une croix permet de fermer le programme en cours et de revenir au menu principal.

Le deuxième avec une disquette permet de sauvegarder les modifications du programme en cours. À noter que les programmes sont sauvegardés automatiquement si l'on ferme le programme en cours avec le premier bouton ou si l'on ferme l'application entièrement.

Le troisième est ici représenté par une flèche montante. L'icône et la description de ce bouton varient en fonction du type de projet ouvert. Son rôle sera toujours d'*exécuter* le programme en cours. Ici il permet de transmettre le programme vers le robot connecté à l'ordinateur.

Les trois boutons qui viennent d'être présentés seront toujours présents. Tous les autres boutons présents dans la barre d'outils sont en liens avec l'édition et l'exécution du programme et dépendent donc du type de projet ouvert.

Le bouton avec une flèche verte vers la droite permet, une fois le programme transmis vers le robot, de le lancer.

Le bouton avec un panneau stop permet d'arrêter le programme en cours sur le robot. Ces deux derniers boutons ne fonctionnent que si le robot est encore connecté à l'ordinateur.

Sur la figure 5, la zone d'édition du programme est désormais présente. Sur la gauche se trouve la *boîte à outils*, elle est séparée en catégories qui contiennent chacune plusieurs blocs différents qui peuvent être assemblés pour former le programme.

Tous les blocs disponibles sont affichés dans la zone d'édition. La couleur des blocs indique la catégorie à laquelle ils appartiennent

Déplacements

- *Avancer/Reculer/Pivoter à gauche/Pivoter à droite à la vitesse Lente/Normale/Rapide* : Ce bloc permet de faire bouger le robot dans une direction choisie à la vitesse choisie. Le robot se déplacera comme indiqué jusqu'à ce qu'un nouveau bloc pour changer ses déplacements soit rencontré.
- *Avancer/Reculer/Pivoter à gauche/Pivoter à droite à la vitesse <nombre>* : Identique au bloc précédent à la différence que la vitesse est cette fois représentée par un nombre entre 0 et 1. 0 étant l'arrêt, et 1 la vitesse maximale. On peut par exemple mettre 0.5 pour une vitesse moyenne
- *Arrêter* : Arrête le robot

Capteurs

- *Le capteur 1/2/3/4 détecte un obstacle très proche/proche/éloigné* : Renvoi 'vrai' si le capteur donc le numéro est indiqué détecte un obstacle plus proche que la distance indiquée. *très proche* correspond à moins de 8 centimètres, *proche* correspond à moins de 20 centimètres et *éloigné* correspond à plus de 20 centimètres
- *Distance du capteur 1/2/3/4* : Retourne la distance en millimètres mesurée par le capteur indiqué

LED

- *Allumer la LED en <couleur>* : Allume la LED de la carte Wipy avec la couleur indiquée. Par défaut une palette est disponible pour choisir une couleur, mais des blocs de couleurs peuvent être emboîtés définir la couleur qui sera allumée
- *Palette de couleur* : Ce bloc est celui qui est présent par défaut dans le bloc *Allumer la LED en ...* Il permet de choisir une couleur parmi une palette
- *Couleur aléatoire* : Ce bloc retourne une couleur aléatoire. Il peut être emboîté dans le bloc *Allumer la LED en ...* pour allumer la LED dans une couleur aléatoire
- *Eteindre la LED* : Eteint la LED

Temps

- *Attendre <nombre> seconde* : Permet d'attendre un temps donné avant de passer au prochain bloc. Le temps est exprimé en secondes et peut être un nombre à virgule, par exemple en mettant 1.5, la temporisation sera d'une seconde et demie. Ce bloc peut être utilisé pour permettre au robot d'avancer pendant un temps donné avant de s'arrêter par exemple, en le plaçant entre les blocs *Avancer à la vitesse ...* et *Arrêter*

Logique

- *Si, sinon si, sinon* : Ce bloc permet de réaliser des actions seulement si certaines conditions sont réunies. La condition nécessaire pour que les actions se réalisent est emboîtée après le *si*, les actions à réaliser sont placées dans la zone *faire*
- *vrai/faux* : Retourne 'vrai' ou 'faux', selon la valeur indiquée
- *pas* : Inverse la valeur du bloc emboîté à droite. Par exemple *pas vrai* est équivalent à 'faux' et *pas faux* est équivalent à 'vrai'

- $\langle \text{valeur} \rangle = / \neq / < / \leq / > / \geq \langle \text{valeur} \rangle$: Retourne vrai si la comparaison indiquée est vraie, et retourne faux sinon
- $\langle \text{valeur} \rangle \text{ et/ou } \langle \text{valeur} \rangle$: Retourne vrai si les deux expressions à gauche et à droite sont vraies. Dans le cas où le *et* est remplacé par *ou*, ce bloc retourne vrai si au moins une des deux expressions est vraie seulement

Boucles

- *Répéter indéfiniment* : Tous les blocs présents à l'intérieur seront répétés à l'infini
- *Répéter tant que/jusqu'à* : Tous les blocs présents à l'intérieur seront répétés tant que la condition indiquée après le *tant que* sera vraie. Dans le cas où le *tant que* est remplacé par *jusqu'à*, les blocs présents à l'intérieur seront répétés jusqu'à ce que la condition indiquée se réalise
- *Répéter $\langle \text{nombre} \rangle$ fois* : Répète les blocs qui se trouvent à l'intérieur le nombre de fois indiqué

Maths

- $\langle \text{nombre} \rangle$: Retourne un nombre
- $\langle \text{valeur} \rangle + / - / * / / \langle \text{valeur} \rangle$: Retourne le résultat de l'opération mathématique entre les deux valeurs

Variables

- $\langle \text{nom de variable} \rangle$: Retourne la valeur de la variable indiquée
- *Ficher $\langle \text{nom de variable} \rangle$ à $\langle \text{expression} \rangle$* : Modifie la valeur de la variable indiquée en lui donnant la valeur de l'expression placée à droite

Une fois le programme écrit, il est possible de l'envoyer vers la carte en cliquant sur le bouton "*Envoyer le programme vers la carte*". Une bulle apparaît alors pour indiquer où en est rendu l'envoi du programme, puis indique lorsque l'envoi est terminé, ou bien si l'envoi a échoué, indique qu'une erreur s'est produite.

1 Description des classes

Diagramme de classes

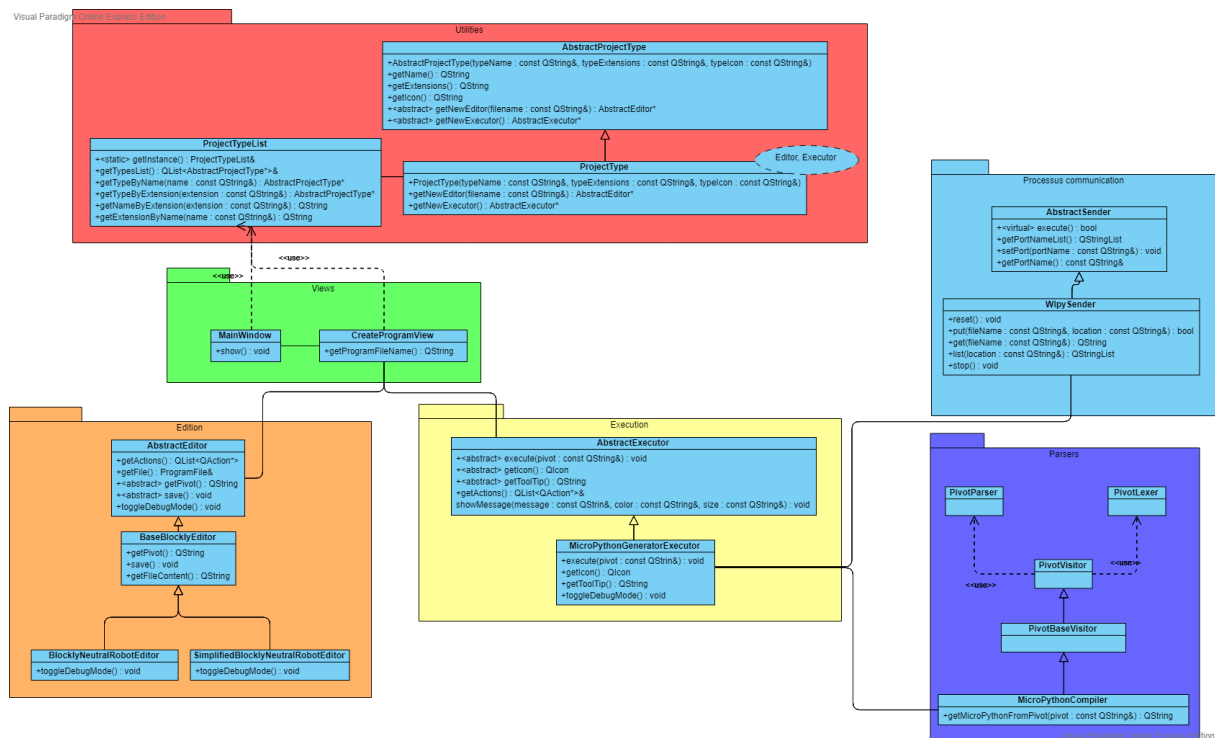


Figure 1 – Diagramme des classes final

Explication des classes

MainWindow : Seules deux vues différentes sont présentes dans l'application. La classe **MainWindow** représente la fenêtre principale de l'application, c'est-à-dire celle qui comprend l'édition des programmes. Son rôle est de contenir l'ensemble des éléments principaux directement à portée de main, comme l'application est destinée à un public jeune, les sous-fenêtres ont été évitées autant que possible. De la même façon, dans la barre d'outils, seules les actions nécessaires à un instant T sont affichées.

La classe **MainWindow** est faite pour embarquer différents éditeurs différents et produire différents comportements au moment de l'exécution. Pour cela, tout ce qui concerne l'édition des programmes et leurs exécutions est placé dans des classes dites "Éditeur" et "Exécuteur" qui seront vues par la suite. À l'ouverture d'un programme, les éditeurs et exécuteurs correspondant sont chargés et intégrés dans l'interface.

La **MainWindow** se charge uniquement de faire communiquer l'éditeur avec l'exécuteur, et de transmettre les actions de base de l'utilisateur à l'éditeur, comme la sauvegarde ou la fermeture du programme en cours.

CreateProgramView : Cette classe représente la vue permettant de créer un nouveau programme. Elle hérite de la classe **QDialog**, elle ne se chargera donc pas de créer le programme elle-même, mais seulement de récupérer le nom du nouveau programme, son emplacement et son type. Ces informations sont alors renvoyées à la fenêtre principale qui se chargera alors de créer et charger le programme.

AbstractEditor : Cette classe représente un éditeur abstrait. Les éditeurs sont les classes qui vont être chargées dans l'interface graphique par la fenêtre principale afin de gérer l'édition des programmes. Tout nouvel éditeur doit hériter de cette classe, et devra implémenter les méthodes abstraites, à savoir :

- **getPivot()**, qui retourne une chaîne de caractère. Cette méthode a pour but de communiquer avec l'exécuteur qui sera associé à l'éditeur. Les éditeurs doivent envoyer sous forme de textes les instructions à effectuer à l'exécuteur. Le lien est fait directement par la fenêtre principale, et ne doit pas être géré directement au sein de l'éditeur. L'éditeur est libre de renvoyer ses instructions sous la forme qu'il veut, il suffit que l'exécuteur associé soit compatible avec le format envoyé, il est possible que plusieurs exécuteurs soient compatibles avec le format d'un éditeur.
- **toggleDebugMode()**, qui est appelée par la fenêtre principale lorsque l'utilisateur demande à activer ou désactiver le mode debug. Cette fonctionnalité est uniquement à destination des développeurs. À noter que pour activer ou désactiver le mode debug, le raccourci est **Ctrl+T**, **Ctrl+D**, **Ctrl+M** successivement (**TDM** pour **Toggle Debug Mode**).

À noter que la classe possède également un attribut protégé appelé **actionsList**. Une action représente un bouton dans la barre d'outils de la fenêtre principale. Au moment de l'intégration de l'éditeur à la fenêtre, la classe **MainWindow** charge les actions présentes dans **actionsList** et les intègre à la barre d'outils. Par défaut cette liste est vide, mais en héritant de **AbstractEditor** vous pouvez modifier cette liste pour intégrer vos propres actions liées à l'éditeur directement dans la barre d'outils.

BaseBlocklyEditor, BlocklyNeutralRobotEditor et SimplifiedBlocklyNeutralRobotEditor : Bien que l'application prévoit de pouvoir recevoir n'importe quel type d'éditeur et d'exécuteur, le but initial était de programmer des robots avec Blockly. Cela implique que plusieurs éditeurs différents utilisant Blockly allaient être créés. La classe `BaseBlocklyEditor` a été mise en place pour simplifier la mise en place d'un éditeur utilisant Blockly. Elle se charge de charger l'interface Blockly, de gérer la sauvegarde et le chargement des fichiers xml contenant les programmes Blockly. Elle implémente aussi la méthode `getPivot()` en retournant la représentation textuelle du programme Blockly, pour qu'il puisse être parsé par l'exécuteur. Les classes héritant de `BaseBlocklyEditor` ont seulement besoin de passer en paramètre au constructeur le chemin du fichier html Blockly qui doit être chargé. `BlocklyNeutralRobotEditor` et `SimplifiedBlocklyNeutralRobotEditor` héritent toutes les deux de `BaseBlocklyEditor` en spécifiant chacune un fichier html différent.

AbstractExecutor : Cette classe représente un exécuteur abstrait. À l'instar d'`AbstractEditor`, les exécuteurs qui sont créés doivent hériter de cette classe. `AbstractExecutor` possède tout comme `AbstractEditor` un attribut protégé `actionsList` pour placer dans la barre d'outils de l'interface des actions personnalisées (voir la description d'`AbstractEditor`).

Cette classe est abstraite et donc certaines méthodes doivent être implémentées :

- `execute(QString)` : Cette méthode sert à recevoir le code envoyé par l'éditeur. L'exécuteur peu alors en faire ce qu'il veut. Il est possible qu'un exécuteur soit compatible avec plusieurs éditeurs différents.
 - `toggleDebugMode()` : Voir description de `AbstractEditor`
 - `getIcon()` : Cette méthode doit retourner l'icone qui représentera l'action permettant l'exécution du programme, elle est utilisée par la fenêtre principale.
 - `getToolTip()` : Tout comme `getIcon()` cette méthode est utilisée par la fenêtre principale, et doit renvoyer le texte descriptif qui représentera l'action permettant l'exécution du programme.
-

MicroPythonExecutor : Cette classe hérite d'`AbstractExecutor` et permet de traduire un programme en langage Blockly sous forme textuel textuel (langage pivot) en MicroPython, et de l'envoyer sur un robot connecté à l'ordinateur. Elle possède également deux actions personnalisées dans la barre d'outils, qui permettent de démarrer le programme sur le robot et de l'arrêter

MicroPythonCompiler : Cette classe est utilisée par `MicroPythonExecutor` pour traduire le langage pivot en MicroPython. Elle possède une méthode `getMicroPythonFromPivot(QString)` pour cela. Elle est le coeur de la logique de l'application et hérite de `PivotBaseVisitor` qui est une classe générée par ANTLR directement à partir de la grammaire. Cela signifie que toute modification de la grammaire peut avoir un impact direct sur les méthodes de cette classe.

AbstractSender : Cette classe sert à définir de façon abstraite un moyen de communication avec les processus extérieurs. Elle possède une méthode principale permettant de lancer un

processus extérieur, ainsi que des méthodes pour récupérer les ports connectés de l'ordinateur et définir sur lequel communiquer.

WipySender : Cette classe hérite de AbstractSender et est utilisée par MicroPythonExecutor pour communiquer avec la carte Wipy du robot. La communication du robot étant gérée par le processus extérieur Ampy, cette classe reprend les commandes d'ampy et les met à disposition via des méthodes (put, get, ls, reset, stop).

AbstractProjectType : Cette classe est utilisée uniquement pour permettre à la classe ProjectTypeList (voir ci dessous) de pouvoir stocker différents types de projets dans un seul conteneur (QList).

ProjectType : Cette classe hérite de AbstractProjectType et représente un type de projet. Un type de projet est défini par une paire éditeur/exécuteur. Cette paire est également caractérisée par une extension de fichier, une icône et une description (titre).

ProjectTypeList : Cette classe est un singleton et stocke tous les différents types de projets définis (liste d'objets ProjectType). Elle possède différentes fonctions permettant de récupérer l'objet ProjectType correspondant à partir de son titre ou son extension, et d'autres fonctions de commodité.

ProgramFile : Cette classe représente un fichier, elle hérite de QFile pour lui ajouter des fonctionnalités permettant une utilisation encore plus simplifiée. Elle possède des méthodes pour modifier le contenu du fichier, récupérer son contenu depuis la classe ou depuis le disque, et enregistrer son contenu sur le disque. Le chemin du fichier est passé en paramètre du constructeur.

Settings : Cette classe est un singleton dédié à enregistrer les paramètres de l'application. Elle hérite de QSettings et propose des getters/setters sur les paramètres directement pour faciliter l'utilisation. À l'heure actuelle seul l'emplacement par défaut des programmes créés est enregistré dans les paramètres de l'application, mais de nouveaux paramètres peuvent être ajoutés facilement.

1.1 Reprise de projet

Installation du projet pour le développement

La première étape pour reprendre le projet est de forker le projet github : <https://github.com/MaxenceRobin/CompilerForRobotic>

Une fois forké, il faut cloner le projet.

Le projet utilise les bibliothèques de Qt pour l'interface graphique. Il est possible de télécharger ces bibliothèques et de les utiliser avec n'importe quel éditeur de code, mais il est vivement conseillé d'utiliser Qt Creator qui intègre par défaut les bibliothèques. La version de Qt Creator doit être au moins 5.xx.

La bibliothèque ANTLR est utilisée, il est nécessaire de télécharger la runtime disponible sur la page <https://www.antlr.org/download.html>, il faut prendre la version pour le C++ en fonction du système d'exploitation sur lequel le projet sera déployé. Il est possible de prendre une configuration différente que celle initiale mais cela n'a pas été testé.

La configuration initiale utilisait Windows 10. Il est également nécessaire que le compilateur utilisé soit le même que celui qui a été utilisé pour compiler la runtime d'ANTLR. Sur le site, il est indiqué que la runtime C++ pour Windows 10 a été compilée avec VS2015, VS2017 fonctionne également grâce à la retrocompatibilité et est le compilateur qui a été utilisé pour développer initialement l'application.

La runtime d'ANTLR pourra être stockée n'importe où sur l'ordinateur, mais il sera nécessaire d'indiquer à Qt Creator que cette bibliothèque est utilisée, dans le fichier .pro.

Il est possible d'ajouter simplement une bibliothèque dans Qt Creator grâce au menu contextuel montré sur la figure 2.

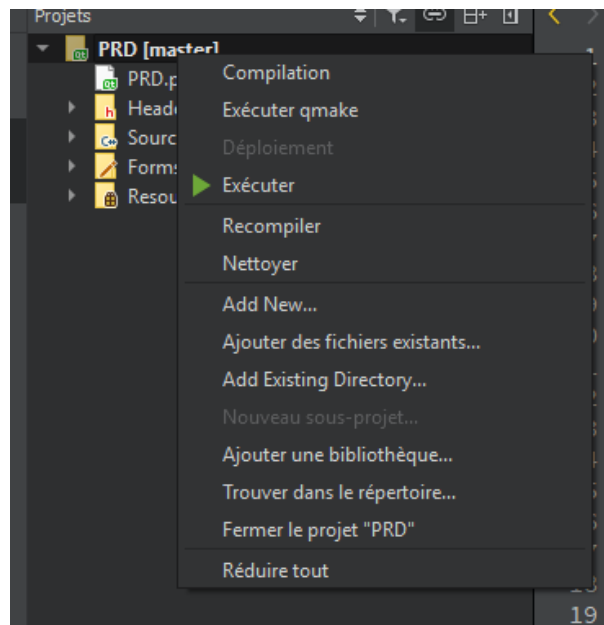


Figure 2 – Menu contextuel pour ajouter une bibliothèque

Une fois cela fait, compiler une première fois le projet pour générer les dossier contenant les exécutables. À l'heure actuelle lancer l'application ne fonctionnerait pas puisque des ressources externes sont nécessaires (Blockly, Ampy...).

Pour cela, créer un dossier nommé exactement "Ressources" dans le dossier où se trouve l'exécutable. Aller ensuite dans le dossier Ressources, et cloner le répertoire git <https://github.com/MaxenceRobin/CompilerForRoboticResources>. L'opération "git clone" créer un répertoire superflu appelé CompilerForRoboticResources, il faut extraire les dossier contenu dans ce dossier et les déplacer directement dans le dossier Ressources.

Il doit désormais y avoir dans le dossier de l'exécutable un dossier "Ressources", qui contient lui-même les dossiers "Ampy", "Blockly", "Generation", "Programs", "Settings".

Il reste une dernière étape qui consiste à cloner Blockly. Comme le répertoire CompilerForRoboticResources contient le dossier Blockly qui contient lui-même le répertoire github du projet Blockly, le clonage de celui-ci doit être fait à part.

Il faut donc se placer dans le dossier "Blockly" et lancer la commande : `git clone https://github.com/google/blockly`

Le sous dossier "Blockly/blockly" devrait maintenant contenir toutes les dernières sources du projet Blockly. À noter qu'il sera possible à tout moment de mettre à jour les sources de Blockly sans toucher au reste du projet en se plaçant dans le dossier "Blockly" et en lançant la commande "git pull".

Vous pouvez maintenant essayer de lancer l'application depuis Qt Creator, toutes les étapes d'installation sont terminées.

Descriptions des méthodes de reprise du projet

Le projet a été conçu de sorte à ce que des fonctionnalités puissent être ajoutées en modifiant le moins possible les classes existantes, mais en ajoutant de nouvelles classes qui héritent des différentes classes abstraites prévues pour cela.

L'architecture de l'application comprend comme point central la classe MainWindow, qui fait le lien entre tous les éléments principaux. Il n'est pas nécessaire de modifier cette classe pour créer un nouveau type de projet, les seuls éléments qui doivent être ajoutés sont un nouvel éditeur et un nouveau contrôleur.

1.1.1 Création de l'éditeur

La première chose à faire pour créer un nouveau type de projet est de créer un nouvel éditeur en créant une classe qui hérite de AbstractEditor. La classe AbstractEditor hérite de QWidget, ce qui signifie qu'elle est un élément de GUI pouvant intégrer différents éléments graphiques. En utilisant l'éditeur QtCreator il est possible de gérer la création d'une fenêtre au cliqué/glissé.

Dans le cas du de la classe existante BaseBlocklyEditor, un élément QWebEngineView est utilisé pour afficher un navigateur web qui se charge de lire les fichiers html en local qui définissent l'interface de Blockly. On pourrait imaginer créer un éditeur permettant de faire la même chose mais en écrivant du code au lieu de manipuler des blocs, en utilisant par exemple un QTextEdit.

En plus du comportement général de l'éditeur, il est nécessaire d'implémenter les méthodes virtuelles pures (abstraites) de AbstractEditor, à savoir getPivot() et toggleDebugMode(). La classe BaseBlocklyEditor n'effectue aucun traitement dans la méthode toggleDebugMode(), mais ce choix est libre pour le développeur.

La méthode getPivot() est la plus importante car elle sert à communiquer avec l'exécuteur sous forme textuelle. La classe BaseBlocklyEditor par exemple converti le programme en blocs sous forme de code dans un langage dit pivot et le retourne. Dans le case de l'éditeur de code

utilisant QTextEdit, la méthode getPivot() pourrait renvoyer directement le programme écrit par l'utilisateur par exemple.

1.1.2 Création de l'exécuteur

La seconde étape consiste à créer l'exécuteur associé à notre éditeur. Il est important de comprendre qu'un éditeur n'est pas forcément lié à un exécuteur en particulier et vice-versa, la seule chose qui fait le lien entre les éditeurs et les exécuteurs est le format des données envoyées par la méthode getPivot() et celui des données reçues par la méthode execute() de AbstractExecutor qu'il va falloir implémenter.

Cette méthode est le coeur de l'exécuteur et définit ce qui devra être fait à chaque fois que l'utilisateur clique sur le bouton pour exécuter le programme.

Dans le cas de MicroPythonExecutor, le format reçu représente le code dans le langage pivot, il est envoyé au MicroPythonCompiler pour générer le code en MicroPython, puis envoyé sur la carte Wipy grâce à la classe WipySender.

On pourrait imaginer vouloir faire la même chose mais pour une carte Arduino, pour cela il faudrait par exemple créer une nouvelle classe ArduinoCompiler capable de gérer le code C++ (Arduino) correspondant, et une classe ArduinoSender qui serait capable d'envoyer le programme vers une carte Arduino. La classe ArduinoExécuteur dans sa méthode execute() aurait alors simplement à appeler tour à tour les classes ArduinoCompiler et ArduinoSender pour transformer et envoyer le programme vers la carte Arduino.

Il sera expliqué plus en détails par la suite comment écrire une classe capable de parser un langage.

Deux autres méthodes doivent être implémenter, getIcon() et getToolTip(). Il suffit de définir ces méthodes en renvoyant l'icône et la description correspondant au rôle de notre exécuteur, pour la classe BaseBlocklyEditor par exemple la description est "Envoyer le programme vers le robot".

1.2 Créer de nouveaux blocs

Dans le cas où l'on voudrait ajouter de nouveaux blocs à l'éditeur de Blockly, il faut suivre les étapes suivantes :

1. Créer le nouveau bloc, à la main ou à l'aide de <https://blockly-demo.appspot.com/static/demos/blockfactory/index.html>
2. Copier la définition du bloc générée et la coller dans le fichier blocks.js qui se trouve dans /Ressources/Blockly/personnal files/neutral robot/ à partir du dossier de l'exécutable.
3. Copier la fonction de génération du code du bloc dans le fichier generator.js qui se trouve au même emplacement que blocks.js
4. Adapter la grammaire (voir suite) pour supporter le nouveau bloc
5. Ajouter le nouveau bloc dans la "boîte à outils" du fichier neutralrobot.html qui se trouve dans /Ressources/Blockly/ à partir du dossier de l'exécutable. (Voir documentation de Blockly pour le fonctionnement de la boîte à outils)

1.3 Parser un langage

L'objectif principal de l'application originale étant de parser un programme écrit avec Blockly pour générer du MicroPython, la bibliothèque ANTLR a été utilisée. La grammaire décrivant le langage pivot retourné par la classe NeutralBlocklyEditor est enregistrée dans le fichier Pivot.g4. Ce fichier comporte des règles lexicales et grammaticales, toute modification de cette grammaire entraînera une re-génération des classes PivotLexer, PivotParser et PivotBaseVisitor dont hérite MicroPythonCompiler.

Par exemple modifier le nom d'une des règles peut modifier le nom d'une des à implémenter dans la classe MicroPythonCompiler.

En cas de modification du langage pivot, il convient donc de modifier dans l'ordre, le fichier Pivot.g4 pour mettre à jour la grammaire puis d'adapter l'implémentation dans MicroPythonCompiler.

Pour plus d'informations sur comment fonctionne ANTLR, se référer directement sur <https://www.antlr.org/>.

1.4 Créer le nouveau type de projet

Maintenant que l'éditeur et l'exécuteur sont définis, il ne reste plus qu'à déclarer le nouveau type de projet pour qu'il soit connu de l'application.

Pour ce faire, il suffit d'ajouter un nouvel objet ProjectType à la liste ProjectTypeList. Pour exemple, les deux types de projets existants sont définis comme montré sur la figure 3

```
ProjectTypeList::ProjectTypeList()
{
    // Creation of the types
    types << new ProjectType<SimplifiedBlocklyNeutralRobotEditor, MicroPythonGeneratorExecutor>
        ("Blockly pour MicroPython simplifié", "sbkmp", ":/icons/blocklyneutralrobot")

    << new ProjectType<BlocklyNeutralRobotEditor, MicroPythonGeneratorExecutor>
        ("Blockly pour MicroPython", "bkmp", ":/icons/blocklyneutralrobot");
}
```

Figure 3 – Déclaration des types de projets dans l'application

La classe ProjectType est une classe template, les deux types à passer en argument sont le type de la classe servant d'éditeur et le type de la classe servant d'exécuteur. Ces deux classes doivent bien entendu hériter de AbstractEditor et AbstractExecutor pour que le code compile.

Le constructeur attend également trois informations qui sont la description du type de projet, l'extension qui sera utilisée pour les fichiers de ce type et le chemin vers l'icône qui sera utilisée pour représenter ce type de projet.

2 Cahier de tests

Tests de MainWindow et CreateProgramView

N°	Description	Statut
1	Création d'un nouveau programme	Non implémenté
2	Ouverture d'un programme	Non implémenté
3	Changement du dossier des programmes par défaut	Non implémenté
4	Fermeture d'un programme	Non implémenté

Tests des éditeurs

N°	Description	Statut
1	Récupération d'un code pivot	Non implémenté
2	Activation/Désactivation du mode debug	Non implémenté
3	Sauvegarde d'un programme	Non implémenté

Tests des exécuteurs

N°	Description	Statut
1	Exécution d'un programme	Non implémenté
2	Récupération de l'icône associée	Non implémenté
3	Récupération de la description associée	Non implémenté

Tests du compilateur

N°	Description	Statut
1	Traduction d'une expression en MicroPython	Non implémenté
2	Traduction d'une instruction en MicroPython	Non implémenté
3	Traduction d'un programme complet	Non implémenté

Tests des sender

N°	Description	Statut
1	Envoi d'un programme vers le robot	Non implémenté
2	Récupération d'un programme à partir du robot	Non implémenté
3	Exécution d'un programme sur le robot	Non implémenté

Tests de classes d'utilités

N°	Description	Statut
1	Ajouter un paramètre au projet	Non implémenté
2	Récupérer un paramètre au projet	Non implémenté
3	Création d'un nouveau type de projet	Non implémenté

G

Glossaire

H

Index



Webographie

- [WWW1] GOOGLE. *Blockly*. URL : <https://developers.google.com/blockly/>.
- [WWW2] Philippe LANGEVIN. *Bison et Flex*. URL : <http://langevin.univ-tln.fr/CDE/LEXYACC/Lex-Yacc.html>.
- [WWW3] QT. *Modules Qt*. URL : <http://doc.qt.io/qt-5/qtmodules.html>.
- [WWW4] Gabriele TOMASSETTI. *ANTLR en C++*. URL : <https://tomassetti.me/getting-started-antlr-cpp/>.
- [WWW5] WIKIPEDIA. *Fonctionnement Bison*. URL : https://fr.wikipedia.org/wiki/GNU_Bison.

Projet Recherche & Développement

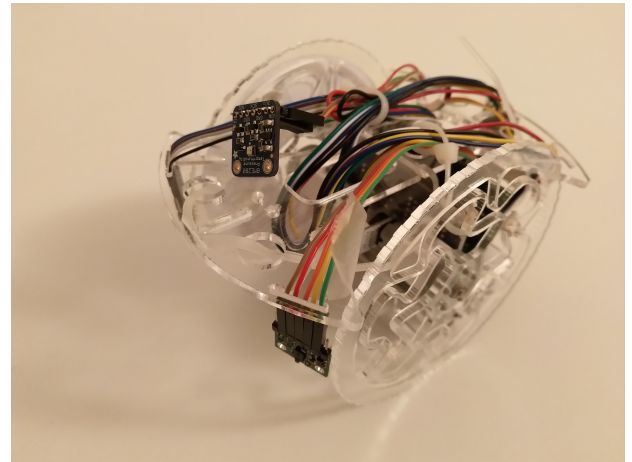
Compilateur pour la robotique

Maxence Robin

Encadrement : Nicolas Monmarche

Contexte

Polytech Tours a mis au point des robots qui peuvent se déplacer en roulant et qui possèdent entre autres des capteurs de température, de luminosité, et de distances pour appréhender leur environnement. Ces robots ont été utilisés jusqu'alors par les élèves de première année pour découvrir la robotique au travers de cours consistant à les programmer.



Robot de Polytech Tours

Problématique

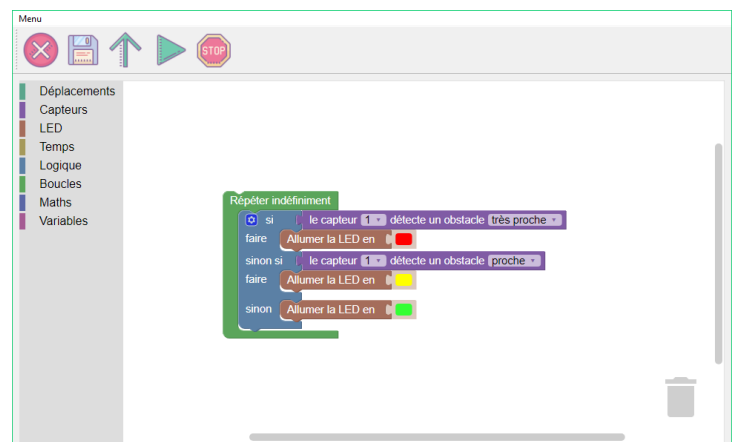
Ces robots pourraient être programmés par des élèves plus jeunes comme des élèves de primaire. Mais les robots sont équipés de cartes qui ne peuvent être programmées qu'en utilisant le langage MicroPython. Il est donc difficile d'apprendre à de jeunes élèves d'utiliser un tel langage de programmation.

```
01 Distance = [-1, -1, -1]
02 Luminosite = [-1.0, -1.0, -1.0]
03
04 d_Thd1 = const(80)
05 d_Thd2 = const(200)
06
07 N_VL6180X = const(3)
08 VL6180X_CE_Pin = ('P6', 'P7', 'P19')
09 VL6180X_I2C_adr_default = const(0x29)
10 VL6180X_I2C_Adr = (const(0x2A), const(0x2B), const(0x2C))
11
12 VL6180X_GPIO_CE_Pin = []
13 for pin in VL6180X_CE_Pin:
14     VL6180X_GPIO_CE_Pin.append(Pin(pin, mode=Pin.OUT))
15     VL6180X_GPIO_CE_Pin[-1].value(0)
16
17 i2c = I2C(0, I2C.MASTER, baudrate=400000)
18 adr = i2c.scan()
19
20 capteur_VL6180X = []
21 for i in range(N_VL6180X):
22     VL6180X_GPIO_CE_Pin[i].value(1)
23     time.sleep(0.002)
24
25     capteur_VL6180X.append(VL6180X(VL6180X_I2C_adr_default, i2c))
26     capteur_VL6180X[i].Modif_Adr_I2C(VL6180X_GPIO_CE_Pin[i], VL6180X_I2C_Adr[i], VL6180X_I2C_adr_default)
27
28 adr = i2c.scan()
```

Exemple de programme en MicroPython

Objectif

L'objectif est donc de développer une application suffisamment simple pour qu'elle puisse être utilisée par des élèves de primaire, et qui permette de programmer ces robots en utilisant à la place du MicroPython, un langage visuel simple utilisant des blocs qui peuvent s'emboîter comme des pièces de puzzle, et qui traduise ensuite le résultat vers du MicroPython pour que le programme puisse être compris par le robot.



Exemple de résultat final attendu pour l'application

Polytech Tours a mis au point des robots qui peuvent se déplacer en roulant et qui possèdent entre autres des capteurs de température, de luminosité, et de distances pour appréhender leur environnement. Ces robots ont été utilisés jusqu'alors par les élèves de première année pour découvrir la robotique au travers de cours consistant à les programmer.



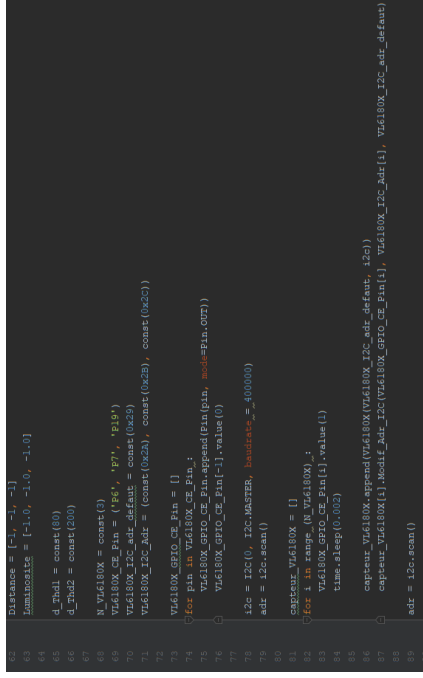
Robot de Polytech Tours

Problématique

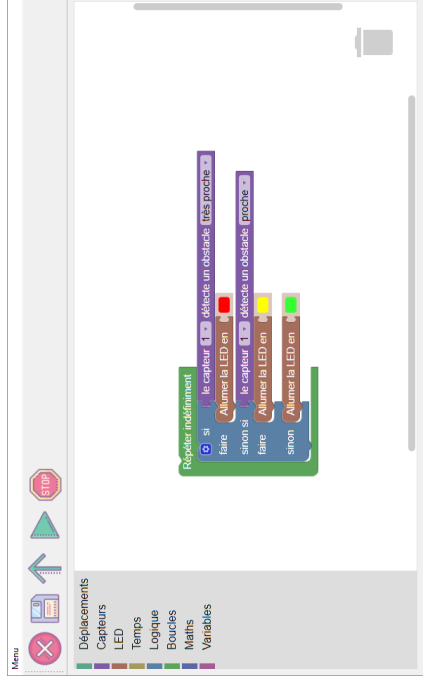
Ces robots pourraient être programmés par des élèves plus jeunes comme des élèves de primaire. Mais les robots sont équipés de cartes qui ne peuvent être programmées qu'en utilisant le langage MicroPython. Il est donc difficile d'appréhender à de jeunes élèves d'utiliser un tel langage de programmation.

Objectif

L'objectif est donc de développer une application suffisamment simple pour qu'elle puisse être utilisée par des élèves de primaire, et qui permette de programmer ces robots en utilisant à la place du MicroPython, un langage visuel simple utilisant des blocs qui peuvent s'emboîter comme des pièces de puzzle, et qui traduise ensuite le résultat vers du MicroPython pour que le programme puisse être compris par le robot.



Exemple de programme en MicroPython



Exemple de résultat final attendu pour l'application

Compilateur pour la robotique

Résumé

Ce projet a pour but de simplifier la programmation de robots en utilisant un langage de programmation visuel sous forme de blocs au lieu d'utiliser un langage de programmation traditionnel. Cette application permet donc la traduction d'un programme visuel vers un langage classique afin de pouvoir le transmettre vers les robots en maintenant la simplicité d'utilisation

Mots-clés

Compilateur, Blockly, MicroPython

Abstract

This project is aimed at simplifying the programming of robots by using a visual programming language, using blocks instead of a traditional coding language. This application allows the translation of a program in a visual programming language into a standard coding language, to upload it into the robot while remaining simple to use

Keywords

Compiler, Blockly, MicroPython