Rapport DSL
Lab 1 - ArduinoML

Équipe D
DE SOUSA VIEIRA Thomas
LACROIX Nicolas
LAMASUTA Mickael
LIAO Chenzhou

Sommaire

Introduction

Équipe

NOM	**Prénom**	**Filière**
DE SOUSA VIEIRA	Thomas	DS4H
LACROIX	Nicolas	FISA
LAMASUTA	Mickael	DS4H
LIAO	Chenzhou	FISE

Description du projet

Ce projet a pour objectif de produire un DSL[1] interne et externe dans le cadre du développement d'un programme Arduino.

D'une durée de 3 semaines, ce projet permet de découvrir le potentiel d'utiliser un DSL, particulièrement lorsque l'utilisateur final n'a aucune connaissance dans la programmation.

Le code du projet est disponible à l'adresse suivante : https://github.com/lm802780/si5-dsl-d.

Dans ce dépôt GitHub, les résultats de chaque scénarios et extensions se trouvent dans le répertoire `results`.

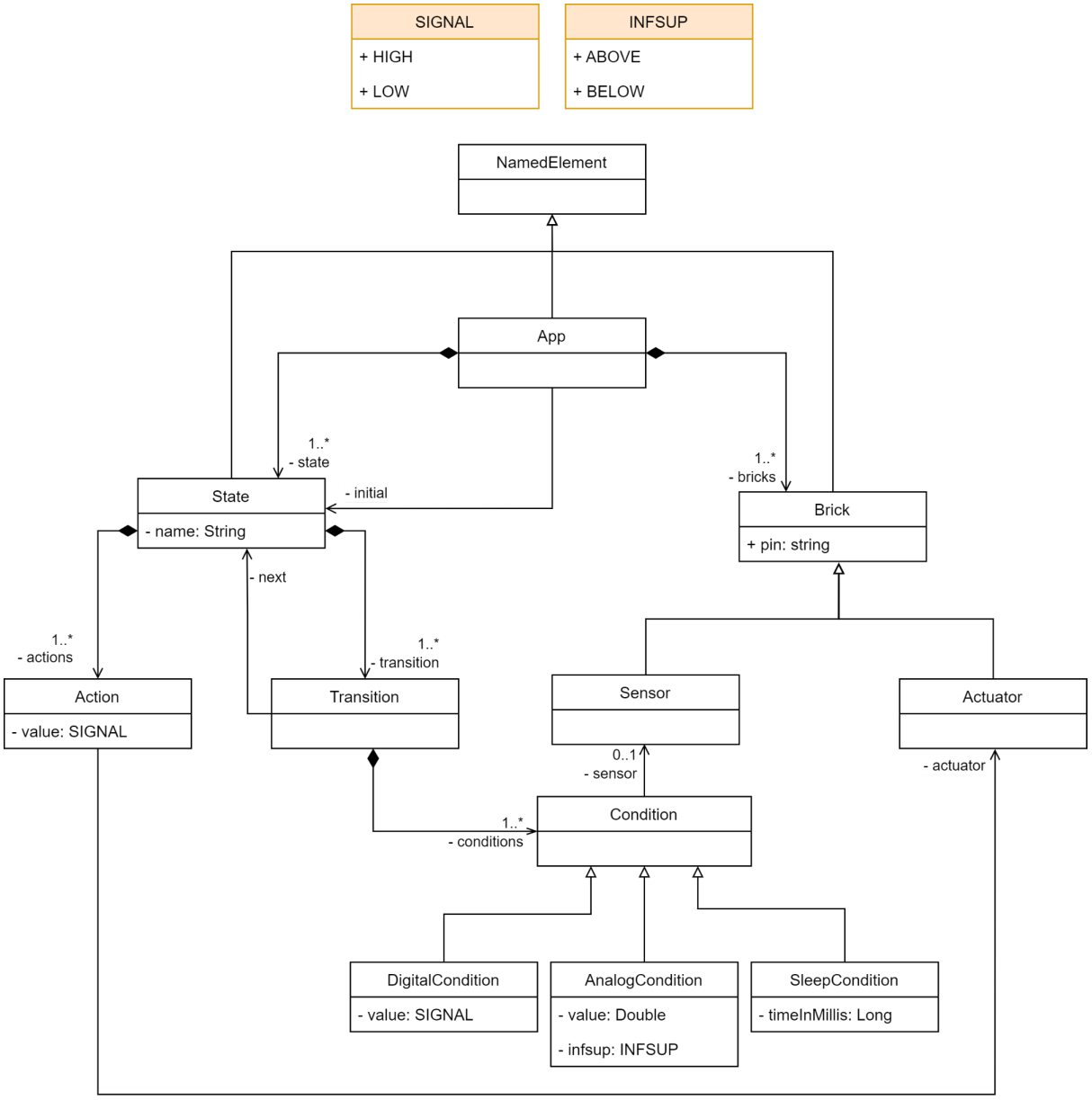
[1] Domain Specific Language : langage utilisé pour répondre à un besoin dans un domaine particulier.

DSL

Kernel

Le sujet du projet propose d'utiliser un kernel disponible à l'adresse suivante : https://github.com/mosser/ArduinoML-kernel.

Nous avons apporté des modifications au kernel fourni afin de pouvoir implémenter les scénarios exigés.



Ce kernel permet de transpiler[2] du code provenant de notre DSL interne et externe. Le kernel produit alors un code compréhensible pour la plateforme Arduino.

[2] Transpiler : traduire du code d'un langage à un autre.

Une transition possède une liste de conditions. Ces conditions sont conjonctives (opérateur logique "et"). Initialement, nous avions prévu d'y inclure l'opérateur logique "ou" mais nous nous sommes rapidement aperçus que cela n'ajoute pas de valeur finale puisqu'il est possible de décomposer un "ou" logique en 2 transitions.

Externe

Nous avons fait le choix d'utiliser le framework ANTLR[3] pour développer notre DSL externe.

Nous avons fait ce choix technologique, car l'équipe sait programmer en Java et le branchement avec le kernel est facilité grâce à Maven.

Voici la grammaire BNF utilisée :

```
<root> ::= <declaration> <bricks> <states>
<declaration> ::= "application " <string> <NEW_LINE>
<bricks> ::= (<sensor> | <analogSensor> | <actuator>)+
<sensor> ::= "sensor " <location>
<analogSensor> ::= "analog sensor " <location>
<actuator> ::= "actuator " <location>
<location> ::= <string> ": " <PORT_NUMBER> <NEW_LINE>
<states> ::= (<state> <NEW_LINE>)*
<state> ::= <initial>? <string> " {" <NEW_LINE> <action>* <transition>* <NEW_LINE> "}"
<action> ::= (<SPACES> <actionable>)+ " <= " <SIGNAL> <NEW_LINE>
<actionable> ::= <string>
<transition> ::= <SPACES> (<digitalTransition> | <analogTransition> | <sleepTransition>)+
<digitalTransition> ::= <condition>+  " => " <string> <NEW_LINE>
<condition> ::= <string> " is " <SIGNAL> <SPACES> (<CONNECTOR> <condition>)?
<analogTransition> ::= <conditionA>+ " => " <string>
<conditionA> ::= <string> <INFSUP> <DOUBLE> <SPACES> (<CONNECTOR> <conditionA>)?
<sleepTransition> ::= <NUMBER> " ms => " <string> <NEW_LINE>
<initial> ::= "-> "

<string> ::= <LOWERCASE> (<LOWERCASE> | <UPPERCASE>)+
<PORT_NUMBER> ::= [1-9] | "11" | "12" | "A0" | "A1"
<SIGNAL> ::= "HIGH" | "LOW"
<CONNECTOR> ::= <SPACES> "AND" <SPACES>
<INFSUP> ::= " ABOVE " | " BELOW "
<NUMBER> ::= [0-9]+
<DOUBLE> ::= ("0" |  [1-9] [0-9]*) ("." [0-9]+ )?
<LOWERCASE> ::= [a-z]
<UPPERCASE> ::= [A-Z]
<NEW_LINE> ::= "\n"*
<SPACES> ::= " "*
```

À noter que cette grammaire ne tient pas compte des commentaires que l'on peut insérer en utilisant "//" en début de ligne. Cela permet de rendre la grammaire plus lisible.

[3] ANother Tool for Language Recognition (https://www.antlr.org/)

La version textuelle est disponible à l'adresse incluse dans l'hyperlien suivant : https://bnfplayground.pauliankline.com/

Interne

Nous avons fait le choix de développer en Groovy notre DSL interne.

Présentés en cours et découvert par certains étudiants de l'équipe il y a quelques années (notamment pour sa capacité à créer un DSL), nous avons souhaité découvrir ce langage. Et, tout comme notre DSL externe avec ANTLR, Groovy est compatible avec Maven, ce qui facilite le branchement avec le kernel.

Voici la grammaire BNF utilisée :

```
<app> ::= <brick_declarations> <state_declarations> <initial> <transition_declarations> <export>
<brick_declarations> ::= (<sensor> | <analog_sensor> | <actuator>)*
<sensor> ::= "sensor" <sensor_name> ("pin" | "onPin") <pin_id> <NEW_LINE>
<analog_sensor> ::= "analog_sensor" <sensor_name> "pin" <analog_pin_id> <NEW_LINE>
<actuator> ::= "actuator" <sensor_name> "pin" <pin_id> <NEW_LINE>
<state_declarations> ::= <state>*
<state> ::= "state" <state_name> ("means" <state_condition>)? <NEW_LINE>
<state_condition> ::=  <sensor_name> "becomes" <state_signal> ( "and" <state_condition>)?
<initial> ::= "initial" <state_name> <NEW_LINE>
<transition_declarations> ::= <transition>*

<transition> ::= "from" <state_name> "to" <state_name> ( "when" <when_closure> | "after" <after_closure>) <NEW_LINE>
<when_closure> ::= <sensor_name> (<digital_condition> | <analog_condition>)
<digital_condition> ::= "becomes" <state_signal> ("and" <when_closure>)?
<analog_condition> ::= ("above" | "below") <analog_value>  ("and" <when_closure>)?
<after_closure> ::= <temporal_condition>
<temporal_condition> ::= <time> "ms"
<export> ::= "export " <script_name> <NEW_LINE>
<sensor_name> ::= <string>
<state_name> ::= <string>
<pin_id> ::= <NUMBER>
<analog_pin_id> ::= <string>
<state_signal> ::= (<space> "low" <space>) | (<space> "high" <space>)
<analog_value> ::= <NUMBER>
<time> ::= <NUMBER>
<script_name> ::= <string>
<string> ::=  <space> "\"" (<LOWERCASE> | <UPPERCASE> | <NUMBER>)+ "\"" <space>
<space> ::= " "*
<LOWERCASE> ::= [a-z]
<UPPERCASE> ::= [A-Z]
<NUMBER> ::= <space> [0-9]+ <space>
<NEW_LINE> ::= "\n"*
```

À noter que cette grammaire ne tient pas compte des commentaires que l'on peut insérer en utilisant "#" en début de ligne. Cela permet de rendre la grammaire plus lisible.

La version textuelle est disponible à l'adresse incluse dans l'hyperlien suivant : https://bnfplayground.pauliankline.com/

Scénarios implémentés

Scénario 1

Pour faciliter l'utilisation de notre DSL par l'utilisateur, nous avons ajouté la possibilité d'ajouter plusieurs actionneurs à la suite. Ainsi, les lignes pour les actionneurs n'ont pas à être dupliquées si leur état final souhaité est le même.

Syntaxe DSL externe :

```
led buzzer <= HIGH
```

Syntaxe DSL interne :

```
state "on" means led becomes high and buzzer becomes high
```

Nous n'avons pas eu besoin de changer le modèle de domaine pour ce scénario.

Scénario 2

Pour implémenter ce scénario, nous avons ajouté l'objet "Condition". Cet objet permet d'enchainer plusieurs conditions à la suite pour ainsi rendre la transition plus évoluée.

Chaque condition contient un SIGNAL qui indique que le capteur doit être à HIGH ou LOW.

Chaque condition entre elles est connectée par la condition AND, nous n'avons pas implémenté de connecteur logique en OR car nous supportons plusieurs transitions à la suite.

Syntaxe DSL externe :

```
buttonOne is HIGH AND buttonTwo is HIGH => on
```

Syntaxe DSL interne :

```
from off to on when buttonTwo becomes "high" and buttonOne becomes
"high"
```

Pour ce scénario nous avons modifié les transitions de manière à qu'elle soit responsable de une à plusieurs conditions.

Scénario 3

Ce scénario n'a pas nécessité de mise à jour du modèle. La syntaxe existante nous a permis de couvrir le scénario.

Scénario 4

Pour ce scénario peu de modifications ont été apportées au DSL. Nous avons ici fait un scénario comprenant plusieurs transitions à la suite.

Scénario "temporal transitions" (extension)

Pour cette extension, nous avons modifié le modèle de manière à supporter plusieurs types de condition. Par héritage de la classe abstraite “Condition”, nous avons ajouté une nouvelle classe nommée “SleepCondition”. Celle-ci contient une valeur représentant un délai en millisecondes. Une fois le délai dépassé, un changement de transition s'effectue.

Syntaxe DSL externe :
```
800 ms => off
```

Syntaxe DSL interne :
```
from "on" to "off" after 800 ms
```

Scénario "handling analogical bricks" (extension)

Pour cette extension, nous avons ajouté un nouveau type de condition appelé “AnalogCondition”. Contrairement à une transition digitale qui ne possède qu’un SIGNAL, la “condition analog” contient une valeur (value) et un “signe” (infsup).

La condition permet donc d'indiquer qu’une valeur d’un capteur doit être supérieure ou inférieure à la valeur de la condition. Ce qui permet ainsi de valider la transition. Tout comme les transitions numériques, plusieurs conditions analogiques peuvent être suivies si l'on souhaite valider l’état de plusieurs capteurs.

Syntaxe DSL externe :
```
temp ABOVE 19 => off
```

Syntaxe DSL interne :
```
from "on" to "off" when "temp" above 19
```

Analyse critique

Extension temporal sleep

Initialement, nous avons identifié cette extension comme une attente active dans un état. Cette attente active, matérialisée par la fonction `delay(800)`, forcerait à rester dans l'état courant pendant 800 millisecondes.

Cependant, nous nous sommes aperçus que cette approche n'était pas ouverte à l'extensibilité dans le cas où l'utilisateur aurait souhaité qu'un événement puisse interrompre cette attente.

Ainsi, nous avons revu l'algorithme d'attente afin de pouvoir éventuellement créer un événement qui interrompt l'attente dans un état.

```
// Horloge globale non initialisée. Permet de retenir l'heure de
la première entrée dans l'état en attente.
long now = 0;

// Entrée dans un état...
      // Initialiser l'horloge lorsque le programme entre dans
      l'état pour la première fois.
      if (now == 0) {
            now = millis();
      }

      // Vérifie si le programme se trouve depuis au moins 800
      millisecondes dans l'état.
      if((millis()-now > 800) ) {
            currentState = nextState;
            now = 0; // Réinitialiser l'horloge globale pour un
                        prochain état qui pourrait utiliser une
                        temporal transition.
            break;   // Break pour éviter de visiter d'autres
                        transitions pour un même état.
      }
```

DSL externe (ANTLR)

Pour les premiers scénarios nous avons essayé de rendre l'écriture plus facile en ajoutant du "confort d'écriture". Avec par exemple, la possibilité d'ajouter plusieurs actionneurs à la suite, d'écrire plusieurs transition successivement ou encore de rendre la transition plus complète avec plusieurs conditions.

Les premiers scénarios nous ont permis de prendre en main ANTLR et le kernel avant d'ajouter de nouvelles fonctionnalités.

Bien que cette syntaxe soit plus simple à écrire, la syntaxe pourrait être améliorée davantage pour aider l'utilisateur. Avec par exemple, la possibilité d'ajouter un "ou" entre les conditions d'une transition pour ne pas avoir à enchaîner plusieurs transitions à la suite.

Il aurait été également intéressant d'améliorer les exceptions levées par les erreurs de syntaxe. Nous indiquons les erreurs de syntaxe en indiquant la ligne et l'erreur sur celle-ci mais des propositions de correction ou d'entrée valide auraient facilité l'utilisation de notre langage par l'utilisateur du domaine.

```
Exception in thread "main" org.antlr.v4.runtime.misc.ParseCancellationException Create breakpoint : line 10:22 token recognition error at: 'T'
```

Enfin, nous n'avons pas mis en place une analyse statique intégré directement à l'IDE qui indiquerait lors de l'écriture du code les incohérences grammaticales. Nous avons fait le choix d'utiliser un vérificateur syntaxique en amont.

Analyse syntaxique

L'un des enjeux inhérents à l'utilisation d'un DSL est la vérification de la bonne utilisation de la syntaxe. Nous avons donc, dans le cadre de ce projet, fait le choix de vérifier la syntaxe du DSL externe (fichiers .arduinoml). En effet, ce dernier ne profite pas de la même aide à la rédaction du script telle que l'autocompletion proposée par les éditeurs de texte supportant le langage Groovy et Java par exemple.

Afin de proposer une description détaillée des erreurs présentes dans le script (numéro de ligne, type d'erreur et valeurs possibles), nous avons écrit un analyseur en Python qui parse le script à la recherche d'erreurs. Le choix du langage se motive principalement par la facilité de manipulation du texte offerte par ce langage ainsi que son aspect interprété facilitant son utilisation (étape de compilation en moins). De plus, il était intéressant pour nous d'employer un autre langage que le Java et Groovy afin de nous confronter à d'autres technologies et d'autres contraintes.

La phase d'analyse se décompose en 3 étapes. La première consiste à vérifier la validité de la déclaration de l'application (en précisant son nom). Celle-ci est assez simple car il s'agit de vérifier la présence du bon mot clé associé à un nom. Afin de compléter cette première étape d'analyse, la présence de lignes vides superflues est signalée dans des warnings afin de renforcer la qualité du script.

```
1
2
3
4 application redButton
5
6 # Declaring bricks
```

```
File: ./exemple/invalid\warnings_unecessary_blank_line.arduinoml
WARNING l.1: Unecessary blank line.
WARNING l.2: Unecessary blank line.
WARNING l.3: Unecessary blank line.
```

La seconde étape consiste à vérifier la déclaration des briques dans le bloc dédié. Pour cela, nous utilisons une expression régulière validant dans un premier temps le format **type nom: valeur** puis en vérifiant que le type donné est bien supporté par le langage. Une fois validés, les différentes briques sont conservées en mémoire par l'analyseur pour la dernière étape.

```
File: ./exemple/invalid\syntax_error.arduinoml
ERROR l.4: Invalid brick sensore. Should be one of the following ('sensor', 'analog sensor', 'actuator', 'buzzer')
```

Enfin, l'analyseur détecte les différents blocs d'états, les compte pour s'assurer qu'ils sont bien tous fermés puis pour chaque bloc, vérifie les changements de valeurs pour les briques (en s'assurant que les briques ont bien été déclarées et que les états sont bien supportés) et changements d'états en vérifiant que l'état existe bien.

```
File: ./exemple/invalid\syntax_error_unclosed_block.arduinoml
WARNING l.12: Uncessary blank line.
ERROR l.4: Invalid block opening token in line 13. Close the block declared in line 8 first.
File: ./exemple/invalid\syntax_error_unknown_state.arduinoml
ERROR l.10: Invalid state UNDEFINED. Should be one of the following: ('LOW', 'HIGH')
```

L'analyseur possède tout de même certaines limitations. La vérification de l'attribution de deux capteurs sur un même pin n'est pas vérifiée par exemple.

Ce script d'analyse est donc un complément à la vérification faite lors de la conversion via la librairie *io.github.mosser.arduinoml* et notamment par les classes *ArduinomlLexer* et *ArduinomlParser*.

Une évolution possible de l'analyseur serait de l'intégrer à un éditeur de code (ex: VSCode) afin de vérifier la syntaxe au fur et à mesure que l'utilisateur rédige le script.

DSL interne (Groovy)

L'un des avantages de Groovy, comme DSL interne, est que nous n'avons pas besoin de redéfinir le modèle et que nous pouvons directement utiliser le kernel Java (via un import). De cette façon, le DSL externe et interne profitent tous deux du même kernel. Ainsi, lorsqu'une modification est appliquée au kernel, les deux DSL profitent de cette modification afin de répondre à nos besoins.
Groovy nous permet de faciliter l'écriture du DSL puisqu'il n'est pas obligatoire de mentionner les parenthèses pour déclarer une méthode (comme en Java par exemple). Le method chaining de Groovy étend cela en nous permettant de chaîner des méthodes sans parenthèses et sans points entre les appels chaînés.

Exemple d'un DSL en Java :

```
state("on").actuator("buzzer").becomes("low")
```

En comparaison à celui de Groovy :

```
state on means led becomes low
```

L'inconvénient de Groovy par rapport aux autres DSL internes est qu'il ne peut pas trouver les erreurs de syntaxe au moment de la compilation ou avant la compilation (IDE). Si on a une phrase de "foo bar", où les méthodes "foo" "bar" ne sont pas implémentées, le compilateur détectera ces erreurs une fois l'exécution du programme.

Répartition du travail

Étudiant	**Tâches réalisées**
DE SOUSA VIEIRA Thomas	● Développement du kernel pour les scénarios 1, 2, 3, 4 et l'extension analogique. ● Rédaction du DSL externe (ANTLR) pour les scénarios 1, 2, 3 et l'extension analogique.
LAMASUTA Mickael	● Développement du kernel pour l'extension temporal transition. ● Rédaction du DSL externe (ANTLR) pour les scénarios 4 et l'extension temporal transition.
LACROIX Nicolas	● Développement du validateur pour le DSL externe (applicable aux fichiers .arduinoml)
LIAO Chenzhou	● Développement du DSL interne (Groovy). ● Refactoring sur le kernel à propos des transitions et des conditions.