Projet de compilation
Travail à remettre le 16 mai 2022

VERSION 2.0 du `2022-04-09`

Modalité pour rendre le travail

— S'inscrire dans un groupe de 4 étudiants sur le site Moodle. Ce projet nécessite le concours de plusieurs personnes pour être réalisé dans le temps imparti. De plus il s'agit d'un exercice de travail en collaboration. Il est donc interdit de le réaliser seul, sauf autorisation express de l'équipe enseignante.

— Pour rendre ce travail avant la date butoir :
Déposer dans le Moodle du cours, dans *Rendu projet* :
`https://moodle1.u-bordeaux.fr/mod/assign/view.php?id=532819`
  — Un fichier d'archive qui contient tout le code (mais aucune bibliothèque ni aucun des fichiers compilés par javac ou gcc)
  — Un fichier PDF très court qui contiendra quelques notes à destination du correcteur pour mieux comprendre et évaluer votre dépôt et qui explique le rôle des 4 personnes dans le groupe de projet.

1 Introduction

Il est question dans ce projet d'écrire un compilateur `arduinoCode` qui traduit le nouveau langage de programmation `arduinoCode` en code assembleur pour le microcontrôleur `ATmega328p` d'Atmel [1] monté sur un `Arduino Uno`.

La motivation de ce projet est la création d'un langage de programmation très simple et exclusivement dédié à l'`Arduino Uno`.

La version dont il sera question ici est une version simplifiée que l'on pourra étendre par la suite.

1.1 Arduino Uno

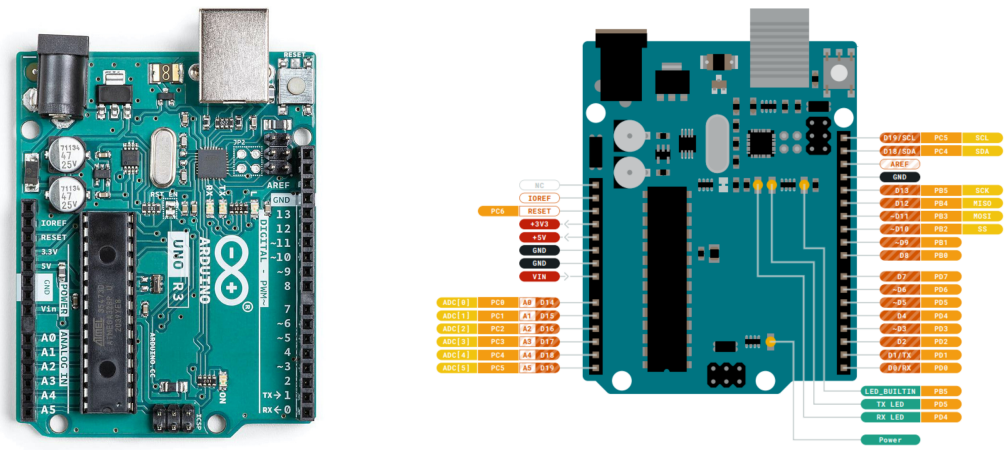


FIGURE 1 – Arduino-uno

1. Maintenant Microchip

Un `Arduino Uno` est une carte électronique qui permet une interface simplifiée avec le micro-contrôleur `ATmega328p`. Elle permet :

— l'alimentation électrique du microcontrôleur,
— la connexion à un ordinateur par le port USB interfacé au port série du micro-contrôleur,
— l'accès simplifié aux entrée-sorties analogiques et digitales du micro-contrôleur,
— une préprogrammation standard du chargeur d'amorçage (*bootloader*, programme lancé à l'allumage de l'appareil).

1.2 ATmega328p

FIGURE 2 – ATmega328p

`ATmega328p` est un micro-contrôleur, c'est-à-dire un circuit intégré qui rassemble un processeur, une mémoire et des interfaces d'entrée-sortie. `ATmega328p` est très utilisé pour la partie de l'informatique embarquée qui demande un minimum de programmation (électroménager, jouets, machines-outils, robotique industrielle, véhicules autonomes, imprimantes, alarmes, etc.)

`ATmega328p` est de la famille AVR Atmel. En voici les principales caractéristiques :

— **Mémoire**
La mémoire est séparée en trois parties :
— **FLASH** 32Ko organisé en mots de 16 bits. C'est l'espace mémoire qui contient le programme et les données constantes (16K × mots de 16 bits). Mémoire non volatile (comme une clef USB). Cet espace contient également le code de démarrage du processeur (*bootloader section*).
— **SRAM** 2Ko organisé en octets. C'est l'espace des registres et aussi l'espace de stockage des variables de notre programme. Mémoire volatile.
— **EEPROM** 1Ko organisé en octets. C'est un espace de stockage de données de configuration. Mémoire non volatile qui sert généralement aux configurations des appareils. Nous ne l'utiliserons pas.

— **Registres**
Il y a 32 registres de travail de 8 bits notés R_0 à R_{31}.
Ces registres sont directement utilisables par le processeur, soit seuls sur 8 bits, soit par couples sur 16 bits.
On y distingue 6 registres particuliers (R_{26} à R_{31}, nommés X, Y, Z sur 16bits) qui permettent un adressage indirect de la mémoire SRAM.
Plusieurs autres registres sont fondamentaux au bon fonctionnement du processeur.
— `SREG`
Ce registre contient un état qui traduit le résultat du dernier calcul. On y trouve par exemple le bit de résultat zéro qui permet de savoir si une comparaison a échoué ou non.
Nous l'utiliserons pour les boucles, les tests, etc.
— `SPH` et `SPL`
Ces deux registres (SP-High et SP-Low) contiennent les adresses sur 16 bits du sommet de la pile. Les instructions `PUSH`, `POP`, `CALL` et `RET` modifient ces registres.
Nous l'utiliserons éventuellement pour le passage par valeur sur la pile des fonctions et aussi pour sauvegarder des données avant des appels qui modifient ces données.
— `DDRB, DDRC, DDRD PINB, PINC, PIND PORTB, PORTC, PORTD`
Ces registres servent à configurer les ports d'entrée-sortie du micro-contrôleur et à envoyer ou à recevoir des données pour commander des périphériques. Nous ne les utiliserons pas directement, car nous allons utiliser des appels à des routines déjà définies.

— **Instructions**
`ATmega328p` contient un jeu restreint d'instructions (RISC). On y distingue les instructions logiques, arithmétiques, de transferts d'information, de sauts et d'appels.
Nous ne détaillons pas le jeu d'instructions ici, nous vous renvoyons à ce document de référence pour trouver une instruction en particulier : `https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf`

1.3 Langage de programmation *arduinoCode*

Pour se faire une première idée, voici un exemple de code écrit dans ce langage :

```
begin
// la pin 13 de l'arduino est en sortie
pinMode(PIN_13, OUTPUT);

loop {
    // affichage sur la sortie série
    puts("Vous avez un message");
    putc('\n');
    // la pin 13 de l'arduino est ON/OFF en alternance
    // délai de 1000ms
    digitalWrite(PIN_13, HIGH);
    delay_ms(1000);
    digitalWrite(PIN_13, LOW);
    delay_ms(1000);
}
end
```

Ce code initialise la broche 13 de l'`Arduino` pour un usage en sortie, puis lance une boucle pour afficher un message et faire clignoter une LED d'essai qui serait branchée sur cette broche.

Nous remarquons qu'il existe un grand nombre de mots clés qui correspondent à des routines prédéfinies (`pinMode`, `puts`, etc). Il en sera question plus loin.

1.4 Langage d'assemblage `AVR Assembler` cible du compilateur

Le langage d'assemblage que nous utilisons est `AVR Assembler` d'Atmel.

Un fichier `AVR Assembler` suit un langage d'assemblage (un langage de programmation de très bas niveau). Il a comme extension `.s` et présente trois sections principales : les données variables, les données constantes et le code :

```
.section .data

; Cette section est réservée aux données variables.
; Elle se trouve dans la mémoire SRAM.

.section .rodata

; Cette section est réservée aux données constantes.
; Elle est inscrite dans la mémoire FLASH lors de la programmation du microcontroleur.

.section .text

; Cette section est réservée au code.
; Elle est inscrite dans la mémoire FLASH lors de la programmation du microcontroleur.

.end
```

Pour se faire une première idée, voici un exemple de code écrit dans ce langage qui reproduit le programme `arduinoCode` précédent :

```
#include <avr/io.h>
.include "data/m328Pdef.s"

.section .rodata

Cst_3:
    .asciz  "Vous␣avez␣un␣message\n"

.section .text
.org 0x00
.include "data/delay-2.s"

.global main_program
main_program:

    push r28
    push r29
    in r28,SPL
    in r29,SPH
    ldi r25, 0
    ldi r24, 13
    ldi r23, 0
    ldi r22, 1
    call pinMode
.LOOP_0:
    ldi r24, lo8(Cst_3)
    ldi r25, hi8(Cst_3)
    call uart_puts
    ldi r25, 0
    ldi r24, 10
    call uart_putc
    ldi r25, 0
    ldi r24, 13
    ldi r23, 0
    ldi r22, 255
    call digitalWrite
    ldi r25, 3
    ldi r24, 232
    call delay_ms
    ldi r25, 0
    ldi r24, 13
    ldi r23, 0
    ldi r22, 0
    call digitalWrite
    ldi r25, 3
    ldi r24, 232
    call delay_ms
    rjmp .LOOP_0        ; eternal loop
.END_LOOP_0:        ; break jump
    pop r29
    pop r28
    ret
.global __do_copy_data
```

```
.end
```

1.5 Une astuce pour « tricher » en regardant comment `avr-gcc` produit du code `AVR Assembler` pour s'en inspirer

On peut faire produire du code `AVR Assembler` directement à partir d'un programme écrit en C en utilisant le compilateur `avr-gcc`. Ceci va nous servir pour examiner du code `AVR Assembler` et s'en inspirer pour écrire la production de code.

Par exemple le code C suivant, enregistré dans `data/essai.c`

```
#include <stdint.h>

uint8_t x;

void main(){
    x = 36;
}
```

produira le code suivant :

```
    .file   "essai.c"
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
    .text
    .comm   x,1,1
.global main
    .type   main, @function
main:
    push r28
    push r29
    in r28,__SP_L__
    in r29,__SP_H__
/* prologue: function */
/* frame size = 0 */
/* stack size = 2 */
.L__stack_usage = 2
    ldi r24,lo8(36)
    sts x,r24
    nop
/* epilogue start */
    pop r29
    pop r28
    ret
    .size   main, .-main
    .ident  "GCC: (Homebrew AVR GCC 9.3.0_3) 9.3.0"
.global __do_clear_bss
```

Il suffit d'observer les deux lignes

```
    ldi r24,lo8(36)
    sts x,r24
```

pour savoir comment on copie une valeur vers une variable d'un octet en `AVR Assembler`.

Pour obtenir un fichier `data/essai.s` à partir d'un fichier `data/essai.c`, il suffit de taper la commande suivante.

```
make data/essai.s
```

1.6 Lien entre langage d'assemblage `AVR Assembler` et routines écrites en langage C

Pour nous permettre de nous concentrer sur la compilation et non sur la partie architecture dans ce projet, nous n'allons pas programmer tout directement en langage `AVR Assembler`, mais nous allons utiliser la bibliothèque développée par la communauté `Arduino`, et notamment `Arduino.h`.

Les routines de cette bibliothèque ont été écrites en langage C ou en langage `AVR Assembler` et l'on trouve les fonctions suivantes :

- `void pinMode(uint8_t pin, uint8_t mode)`
  Programme la broche `pin` à être en sortie (`OUTPUT`), en entrée (`INPUT`), ou en entrée avec une résistance de tirage (`INPUTPULLUP`).
- `void digitalWrite(uint8_t pin, uint8_t val)`
  Envoie un signal (`HIGH` ou `LOW`) sur la broche `pin`.
- `int digitalRead(uint8_t pin)`
  Lit un signal sur la broche `pin`.
- `int analogRead(uint8_t pin)`
  Lit un signal sur la broche `pin`.
- `void analogReference(uint8_t mode)`
  Fixe une référence de signal.
- `void analogWrite(uint8_t pin, int val)`
  Envoie un signal analogique sur la broche `pin`.
- `uint16_t getc(void)`
  Reçoit depuis le port série.
- `uint16_t peek(void)`
  Reçoit depuis le port série sans détruire.
- `void putc(uint8_t data)`
  Émet un octet sur le port série.
- `void puts(const char *s)`
  Émet une chaine de caratères sur le port série.
- `uint16_t available(void)`
  Indique que le port série est prêt à recevoir.
- `void flush(void)`
  Supprime l'entrée du port série.
- `void delay_ms(uint16_t milliseconds)`
  Provoque un délai de plusieurs millisecondes.
- `void delay_s(uint16_t seconds)`
  Provoque un délai pendant plusieurs secondes.

Pour lier le programme écrit en assembleur et le programme écrit en langage C, il suffira de compiler les deux et de créer un lien avec la commande `ld`. Nous avons prévu ceci dans fichier `Makefile` livré avec le projet source, et vous n'avez pas à le faire.

Comment utiliser une fonction C dans le code AVR Assembler ?

1. Passer les paramètres sous la forme d'une liste d'octets à partir du registre `r25`.
2. Appeler l'instruction `call`.

La liste des registres est utilisée comme suit pour chaque argument, en décalant pour chacun la taille de l'argument qui précède :

- Un argument d'un octet est passé à `r24`.
- Un argument de 16bits est passé à `r25:r24`.
- Un argument de 32bits est passé à `r25:r22`.
- Un argument de 64bits est passé à `r25:r18`.

Si la taille de la somme des arguments dépasse 64 octets, la pile est utilisée.
Exemple :

Soit la fonction implémentée en C `void pinMode(uint8_t pin, uint8_t mode);`.

Le code assembleur permettant d'appeler `pinMode(13, 1)` est le suivant :

```
ldi r25, 0
ldi r24, 13
ldi r23, 0
ldi r22, 1
call pinMode
```

1.7 Fonctionnement du programme `arduinoCode`

Le compilateur `arduinoCode` va lire en une seule passe un fichier et va produire le code en langage `AVR Assembler` directement sur sa sortie standard en cas de succès.

Il produira sur sa sortie erreur un document permettant de déboguer : tokens lus, règles de syntaxe réduites, *visitors* analysés, erreurs de syntaxe, erreurs de typage, erreurs sémantiques.

Par convention, les extensions de fichier suivantes ont été choisies :

`.arduinoCode`	code écrit en langage `arduinoCode`
`.log`	fichier de débogage
`.s`	code en `AVR Assembler`

1.8 Étapes de compilation d'un projet `arduinoCode`

Pour simplifier cette étape, nous avons créé un fichier `Makefile` qui produit l'ensemble des étapes automatiquement

Par exemple la commande `make data/test-2.fuse` produira le code suivant :

1. Production de `data/test-2.s` (code assembleur) et `data/test-2.log` (messages de débogage) à partir de `make data/test-2.arduinoCode`.

```
java -cp classes fr.ubordeaux.arduinoCode.Main data/test-2.arduinoCode
> data/test-2.s 2> data/test-2.log
```

2. Production du fichier objet `data/test-2.o` à partir du programme `AVR assembler` `data/test-2.s`.

```
avr-as -mmcu=atmega328p -o data/test-2.o data/test-2.s
```

3. Production du code exécutable `data/test-2.elf` à partir de tous les codes objet `*.o`.

```
avr-ld -m avr5 -o data/test-2.elf data/test-2.o
lib/ArduinoCore-avr-master/cores/arduino/wiring_digital.o
lib/ArduinoCore-avr-master/cores/arduino/wiring_analog.o
lib/ArduinoCore-avr-master/cores/arduino/wiring.o
lib/ArduinoCore-avr-master/cores/arduino/hooks.o
lib/avr-uart-master/uart.o    src/arduinoCodeMain.o
```

4. Production du code en format hexadécimal `data/test-2.hex` destiné à être flashé sur un microcontrôleur.

```
avr-objcopy -O ihex -R -eeprom data/test-2.elf data/test-2.hex
```

5. Enfin, pour téléverser le programme `data/test-2.hex` sur l'`Arduino Uno` connecté au port série il faut simuler la création du fichier `data/test-2.fuse`.

```
avrdude -p atmega328p -c arduino -P /dev/cu.usbserial-1410 -b115200 -D
-U flash:w:data/test-2.hex:i > data/test-2.fuse
```

1.9 Fonctionnement des *visitors*

La première étape du compilateur consiste à créer un arbre de syntaxe abstrait (AST) sur les déclarations, les expressions et les instructions et à lancer la méthode `void accept(Visitor visitor)` sur chacun d'entre eux.

Ces appels à `accept(Visitor visitor)` vont avoir comme effet de faire le traitement demandé par `visitor` sur chaque AST.

Un `visitor` est un objet qui contient la méthode `visit` dont le code dépend de son argument (elle est surchargée).

Elle a pour but de réaliser une tâche spécifique à cet argument (l'AST en ce qui nous concerne). Par exemple, faire une vérification de type sur une expression, ou de produire le code `AVR Assembler` correspondant à une instruction ou encore d'écrire les déclarations en code `AVR Assembler` des variables et constantes.

Prenons un exemple.

Dans `Parser.y`, on trouve cette ligne :

```
stm:
    ...
    | left_part '=' expression ';' { $$ = new StmAFF($1, $3); }
```

Elle a pour effet de construire un AST `StmAff` pour une instruction (`stm`) affectation.

Plus haut dans le code, on trouve ceci :

```
list_of_stms:
    stm {
      try{
        $1.accept(checkTypeVisitor);
      } catch (TypeException e) {
        System.err.println(e.getMessage());
        return YYERROR;
      }
...
    }
...
```

Ceci a pour effet de lancer un *visitor* de type `CheckTypeVisitor` sur l'AST `$1` et de traiter l'exception si ce traitement échoue.

Or dans le code de la classe `CheckTypeVisitor`, on trouve ceci :

```
    @Override
    public void visit(StmAFF stm) throws TypeException {
        System.err.println("*** visit(Stm) with " + this);
        if (!stm.getLeft().getType().equivalent(stm.getRight().getType()))
            throw new TypeException("types should be equivalent");
    }
```

Ce code va donc lancer une exception si les types de la partie gauche et droite de l'affectation ne sont pas équivalents.

Il y aura dans `CheckTypeVisitor` une implémentation de la méthode `visit` pour chaque AST dont on souhaite vérifier les types (`StmIF`, `StmWHILE`, `StmFOREACH`, `ExprBinary`, etc.)

Le principe est le même pour le `visitor` de type `CodeGeneratorVisitor` qui produit le code `AVR Assembler` directement en sortie standard du programme.

1.10 Fichiers livrés

Ce projet contient de nombreux fichiers. Nous allons en expliquer l'usage

Vous ne devrez normalement intervenir que sur les fichiers annotés par `(***)` et notés en vert sur le schéma.

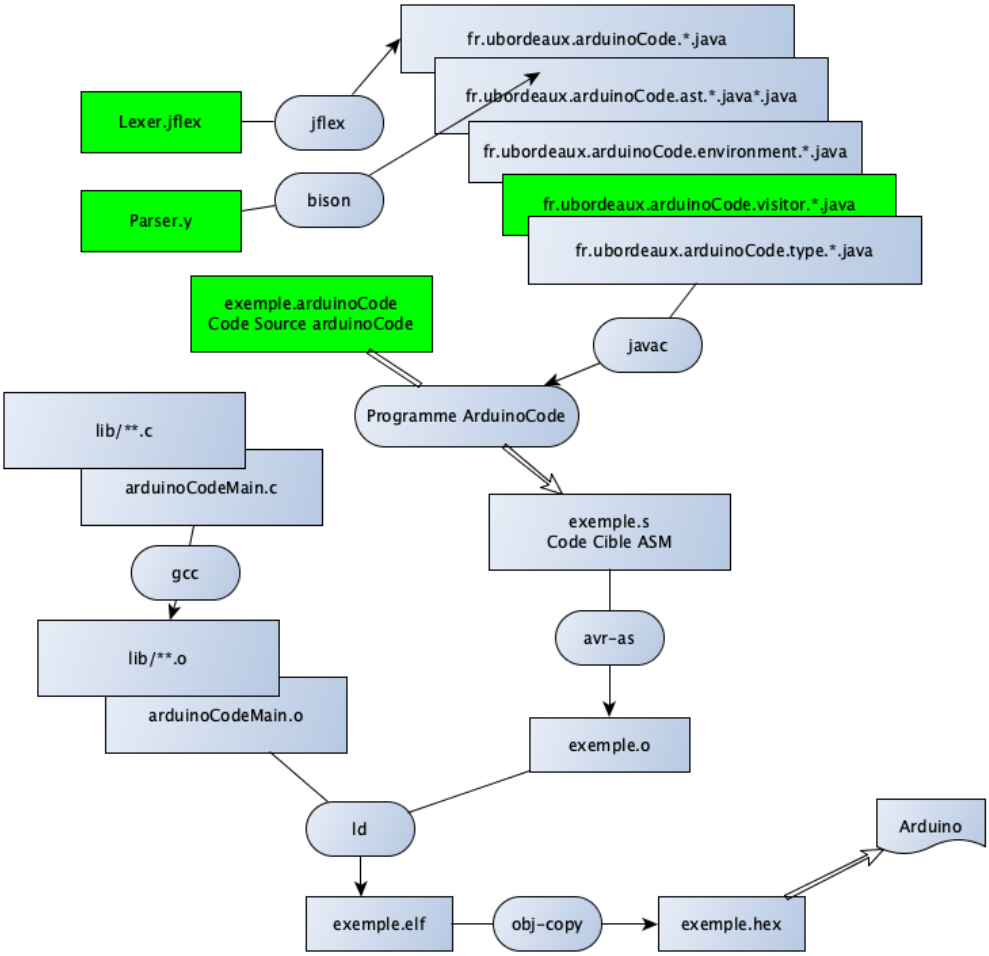


FIGURE 3 – schéma de compilation

1. Fichiers du compilateur `arduinoCode`
   - Analyseur lexical
     `arduinoCode/lexer/Lexer.jflex (***)`
     Ce fichier contient l'analyse lexicale d'`arduinoCode`.
   - Analyseur syntaxique
     `arduinoCode/parser/Parser.y (***)`
     Ce fichier contient l'analyse syntaxique d'`arduinoCode`. Il produit des arbres de syntaxe abstrait et lance des *visitor* dessus pour a) vérifier le type des variables et des fonctions, b) produire le code assembleur.
   - Programme principal `arduinoCode`
     `fr.ubordeaux.arduinoCode.Main.java`
     Ce programme ouvre un fichier donné en argument et lance l'analyse syntaxique.
   - Représentation de l'arbre de syntaxe abstrait
     `fr.ubordeaux.arduinoCode.ast.*`
     Ces classes permettent de construire des objets complexes à partir de l'analyse de la structure d'un fichier `arduinoCode`.
   - Table des symboles
     `fr.ubordeaux.arduinoCode.environment.*`
     Ces classes permettent de conserver les définitions des variables, des types, des fonctions, etc. lors de l'analyse syntaxique.
   - Représentation des expressions de type
     `fr.ubordeaux.arduinoCode.type.*`
     Ces classes permettent de construire des expressions de type sous forme arborescente.
   - Visiteurs (code lancé automatiquement sur l'arbre de syntaxe abstrait)

`fr.ubordeaux.arduinoCode.visitor`
Ces classes contiennent des méthodes largement surchargées `public void visit(X a) throws Exception` où `X` est une implémentation de `fr.ubordeaux.arduinoCode.ast.Ast`.
Les méthodes sont automatiquement lancées lors de l'analyse syntaxique et il convient que vous interveniez principalement sur ces méthodes pour modifier le projet.
- Le *visitor* `fr.ubordeaux.arduinoCode.visitor.CheckTypeVisitor (***)` réalise des tests sur les types.
- Le *visitor* `fr.ubordeaux.arduinoCode.visitor.CodeGeneratorVisitor (***)` réalise la production de code assembleur directement à partir des arbres de syntaxe abstraits.
- Le *visitor* `fr.ubordeaux.arduinoCode.visitor.DataGeneratorVisitor (***)` réalise la production de code assembleur pour la déclaration des variables et la définition des constantes complexes directement à partir des arbres de syntaxe abstraits.

2. Fichiers d'un projet `arduinoCode`
   - `data/*.arduinoCode`
     C'est un fichier édité par l'utilisateur du projet pour programmer un `Arduino Uno`. Il respecte la syntaxe dictée par l'analyseur lexical et l'analyseur syntaxique.
   - `data/*.s`
     C'est le fichier produit par l'application à partir d'un fichier `data/*.arduinoCode`. Il s'agit d'un fichier en langage `AVR Assembler` qui contient le code pour programmer un `Arduino Uno`.
   - `data/m328Pdef.s`
     Fichier en langage `AVR Assembler` qui contient des définitions pour l'`Arduino Uno`.
   - `src/arduinoCodeMain.c`
     Fichier en langage `C` qui permet de lier le code écrit en assembleur avec des routines plus simplement écrites en C (communication avec le port série).
3. Fichiers pour la compilation
   - `./build.xml`
     Permet de compiler le projet `arduinoCode`.
     - `ant clean`
       Supprime les fichiers produits.
     - `ant parser`
       Construit l'analyseur syntaxique.
     - `ant lexer`
       Construit l'analyseur lexical.
     - `ant classes`
       Compile les classes Java.

     `ant clean parser lexer classes`
   - `./Makefile`
     Permet de lancer le programme sur des exemples, de compiler le résultat produit et de le téléverser sur un Arduino connecté au port USB.
     - `make data/*.s`
       Construit un fichier en `AVR Assembler` `data/*.s` à partir d'un fichier `data/*.arduinoCode`.
     - `make data/*.fuse`
       Téléverse un fichier compilé de `data/*.s` sur un `Arduino Uno` branché sur le port USB.
       La désignation exacte du port selon l'ordinateur utilisé est indiqué dans `Makefile` à la ligne 3 : `USB = /dev/<votre device USB>`
4. Bibliothèques
   - `lib/AduinoCore-avr-master`
     Ce dépôt contient des sources et des fichiers de configuration pour nombre de cartes Arduino. Nous l'utilisons pour exploiter des routines écrites en code C pour l'`Arduino Uno`.
   - `lib/avr-uart-master`
     Ce dépôt contient des sources en code C pour communiquer avec le port série de l'`Arduino Uno`. Nous aurions pu écrire ce code en assembleur, mais l'usage de cette bibliothèque est un gain de temps.

2 Travail à réaliser en groupes de 4

1. Ajouter la possibilité d'écrire des nombres en hexadécimal et en binaire dans le langage.
   Les nombres hexadécimaux s'écrivent en les faisant précéder de `0x`.
   Exemple : `0x10AB`
   Les nombres binaires s'écrivent en les faisant précéder de `B`.
   Exemple : `B01010000`
   Nous vérifions que ces nombres sont cohérents avec le typage du programme.
2. Ajouter l'une des trois structures de contrôle choisies parmi
   — `if`
   — `while`
   — `foreach`
   Ce choix est plus ambitieux car il suppose l'implémentation des listes, intervalles et ensembles. On implémentera l'une des trois structures, puis on passera aux autres éventuellement

   (a) Mot clefs `if` et `else` du langage `arduinoCode` sont destinés à écrire des instructions conditionnelles tel que dans les exemples suivants :

```
val: integer;

begin
  if (val < 0) {
    puts("negative");
  }
end
```

```
val: integer;

begin
  if (val < 0) {
    puts("negative");
  } else {
    puts("positive");
  }
end
```

```
val: integer;

begin
  if (val < 0) {
    puts("negative");
  } else if (val == 0) {
    puts("null");
  } else {
    puts("positive");
  }
end
```

   i. Ajouter une classe `StmIF` dans le package `fr.ubordeaux.arduinoCode.ast` qui permet de représenter un arbre de syntaxe abstrait pour cette nouvelle structure de contrôle.

   ii. Modifier `fr.ubordeaux.arduinoCode.visitor.CheckTypeVisitor`. Ajouter une méthode `public void visit(StmIF stmIF) throws Exception` pour vérifier si les expressions conditionnelles dans cette nouvelle structure de contrôle sont bien des expressions booléennes.

   iii. Modifier `fr.ubordeaux.arduinoCode.visitor.DataGeneratorVisitor`. Ajouter une méthode `public void visit(StmIF stmIF) throws Exception` pour produire le code `AVR assembler` qui correspond aux déclarations de variables et de constantes contenues dans la structure de contrôle.

   iv. Modifier `fr.ubordeaux.arduinoCode.visitor.CodeGeneratorVisitor`. Ajouter une méthode `public void visit(StmIF stmIF) throws Exception` pour produire le code `AVR assembler` de cette nouvelle structure de contrôle.
   Pour réaliser en code `AVR assembler` le code qui correspond à

```
if (<TEST>)
   <STM1>
else
   <STM2>
```

   nous écrivons :

```
    ;; <Code de la partie <TEST>
    ;;  dont le résultat (zero si faux)
    ;;  est dans un registre (par exemple r4)
    tst r24 ; - Test for Zero or Minus
    breq .L2 ; - Branch if Equal
    ;;  Code de la partie <STM1>
```

```
    rjmp .L4 ; - Relative Jump
.L2:
    ;;  Code de la partie <STM2>
.L4:
```

(b) Mot clef `while` du le langage `arduinoCode` est destiné à un usage permettant d'écrire une boucle telle que dans l'exemple suivant :

```
i: integer;
L: list of string;

begin
  i = 65;
  while (i < 65+24) {
    putc(i);
    i++;
  }
end
```

i. Ajouter une classe `StmWHILE` dans le package `fr.ubordeaux.arduinoCode.ast` qui permet de représenter un arbre de syntaxe abstrait pour cette nouvelle structure de contrôle.

ii. Modifier `fr.ubordeaux.arduinoCode.visitor.CheckTypeVisitor`. Ajouter une méthode `public void visit(StmWHILE stmWHILE) throws Exception` pour vérifier si l'expression coditionnelle dans cette nouvelle structure de contrôle est bien une expression booléenne, c'est-à-dire s'il est possible de l'évaluer en faux (0) ou vrai (toute valeur entière non nulle).

iii. Modifier `fr.ubordeaux.arduinoCode.visitor.DataGeneratorVisitor`. Ajouter une méthode `public void visit(StmWHILE stmWHILE) throws Exception` pour produire le code `AVR assembler` qui correspond aux déclarations de variables et de constantes contenues dans la structure de contrôle.

iv. Modifier `fr.ubordeaux.arduinoCode.visitor.CodeGeneratorVisitor`. Ajouter une méthode `public void visit(StmWHILE stmWHILE) throws Exception` pour produire le code `AVR assembler` de cette nouvelle structure de contrôle.

Pour réaliser en code `AVR assembler` le code qui correspond à

```
while (<TEST>)
   <STM>
```

nous écrivons :

```
    rjmp .L2:
.L3:
    ;;  Code de la partie <STM>
.L2:
    ;;  <Code de la partie <TEST>
    ;;  dont le résultat (zero si faux)
    ;;  est dans un registre (par exemple r4)
    tst r24 ;  - Test for Zero or Minus
    brne .L3 ; - Branch if Not Equal
```

(c) Mot clef `foreach` du langage `arduinoCode` est destiné à un usage permettant de parcourir une liste, un ensemble, ou un intervalle comme dans l'exemple suivant :

```
L: list of string;
S: set of string;
R: range of string;

begin
```

```
  x : string;
  L = ["A", "B", "C"];
  foreach x in L {
    puts(x);
  }
  S = {"A", "B", "C"};
  foreach x in S {
    puts(x);
  }
  R = ["A" .. "C"];
  foreach x in R {
    puts(x);
  }
end
```

i. Ajouter une classe `StmFOREACH` dans le package `fr.ubordeaux.arduinoCode.ast` qui permet de représenter un arbre de syntaxe abstrait pour cette nouvelle structure de contrôle.

ii. Modifier `fr.ubordeaux.arduinoCode.visitor.CheckTypeVisitor`.

Ajouter une méthode `public void visit(StmFOREACH stmFOREACH) throws Exception` pour vérifier les types associés à cette nouvelle structure de contrôle.

iii. Modifier `fr.ubordeaux.arduinoCode.visitor.DataGeneratorVisitor`.

Ajouter une méthode `public void visit(StmFOREACH stmFOREACH) throws Exception` pour produire le code `AVR assembler` qui correspond aux déclarations de variables et de constantes contenues dans la structure de contrôle.

iv. Modifier `fr.ubordeaux.arduinoCode.visitor.CodeGeneratorVisitor`.

Ajouter une méthode `public void visit(StmFOREACH stmFOREACH) throws Exception` pour produire le code `AVR assembler` de cette nouvelle structure de contrôle.

Pour réaliser en code `AVR assembler` le code qui correspond à

```
foreach x in [<MIN> .. <MAX>]
   <STM>
```

il faut au préalable préparer l'utilisation de la pile en début du programme.

```
    ;; On met dans Y (r28 :r29) l'adresse de la pile
    in r28,SPL ; Partie basse de l'adresse du sommet de la pile
    in r29,SPH ; Partie haute de l'adresse du sommet de la pile
```

Ensuite on utilisera le sommet de la pile (`Y+1`) comme variable `x`.

```
    ldi r24,<MIN>
    std Y+1,r24
    rjmp .L2
.L3:
    ;; code de <STM>
    ;; où la variable x sera Y+1

    ;; On incrémente x (en retranchant 255)
    ldd r24,Y+1
    subi r24,0xFF
    std Y+1,r24

.L2:
    ;; On teste si égale à <MAX>
    ;; on boucle sur .L3
    ldd r24,Y+1
```

```
    cpi r24,<MAX>
    brne .L3
```

3. Ajouter le code produit par la déclaration de fonctions et de procédures et leur usage dans le langage `ArduinoDode`.

   Pour réaliser en code `AVR assembler` le code qui correspond à

```
procedure foo (arg : <type>) {
   <STMS>
}
```

   nous écrivons :

```
foo:
    push r28
    push r29
    push __tmp_reg__
    in r28,SPL
    in r29,SPH
    ;; <Code de la fonction ici>
    pop __tmp_reg__
    pop r29
    pop r28
    ret
```

   Les arguments sont passés aux registres `r25` à `r18` (cf. plus haut).

   Pour aller plus loin, on pourra s'interroger sur la possibilité que la fonction soit appelée récursivement et puisse contenir des variables locales.

4. Ajouter le code produit par la déclaration de listes (qui sont l'équivalents de tableaux) et leur usage dans le langage `ArduinoDode`.

   Pour réaliser en code `AVR assembler` qui correspond à

```
t: list of integer;

begin
    t[8] = 100;
end
```

   nous écrivons dans la partie données SRAM :

```
    .comm    t,512
```

   et dans la partie code :

```
    ;; Mettre la valeur 100 dans des registres sur 32bits
    ldi r24,100
    ldi r25,0
    ldi r26,0
    ldi r27,0
    ;; Mettre le contenu de ces registres dans t[8],
    ;; c'est-à-dire à l'adresse t + 32
    sts x+32,r24
    sts x+32+1,r25
    sts x+32+2,r26
    sts x+32+3,r27
```

   Pour aller plus loin, on pourra s'interroger sur la possibilité que les listes soient à plusieurs dimensions ou soient typées avec d'autres types complexes (par exemple des intervalles, des ensembles ou des structures).

Annexes

3 Annexe : Mots clefs du langage *arduinoCode*

— **Types**

Mots	Usage	Taille en octets
`pin`	type simple désignant une broche	1
`boolean`	type simple valant `TRUE` ou `FALSE`	1
`byte`	nombre signé de -128 à 127	1
`unsigned byte`	nombre non signé de 0 à 255	1
`small`	nombre signé de –32768 à 32767	2
`unsigned small`	nombre non signé de 0 à 65535	2
`integer`	nombre signé de –2147483648 à 2147483647	4
`unsigned integer`	nombre non signé de 0 à 4294967295	4
`float`	nombre signé à virgule flottante	4
`string[K]`	chaine de caractères	K + 1
`range of T`	intervalle	2
`set of T`	ensemble	2
`list of T`	liste	2
`struct`	structure d'enregistrement	taille des champs inclus
`enum`	type énuméré	1

— **Constantes**

Mots	Type	Usage
`TRUE`, `FALSE`	`boolean`	valeurs booléennes
`PIN_`xx	`pin`	Broche où xx vaut `0` à `13` pour les broches digitales et `A0` à `A5` pour les broches analogiques
`HIGH`, `LOW`	`byte`	signaux haut et bas envoyés ou reçus d'une broche digitale

— **Structures de contrôle**

Mots	Usage	Exemple
`if`, `else`	test	`if (x<100) x++;`
`switch`, `case`, `default`	sauts conditionnels. Remarque : il n'y a pas de `break` dans les `case` et plusieurs opérandes peuvent être supportées par un `case`	`switch (C) { case 12, 13: x++; case 7: x+=2;default: x=0;}`
`foreach`, `in`	boucle sur les éléments d'un intervalle, d'une liste ou d'un ensemble	`foreach x in [1, 2, 3] {y += x;}`
`while`	boucle tant qu'une expression est vérifiée	`while(x<100) x++;`
`do`	boucle jusqu'à ce qu'une expression ne soit plus vérifiée	`do x++; while(x<100);`
`loop`	boucle indéfiniment, tant que break n'est pas rencontré	`loop {x++; if(x>=100)break;}`
`break`	sort de la dernière boucle immédiate	`while (x++ != 0) {if (x>100) break; x++}`
`continue`	passe au cycle suivant de le dernière boucle	`while(x++<100) {if (x%2==1) continue; oddProc(x);}`

— **Opérateurs sur des types complexes**

Mots clef	Types concernés	Usage
`length(x)`	string, set, list, range	donne la longueur de la variable `x`
`first(x)`, `last(x)`	string, list, range	donne le premier élément ou le dernier élément d'une liste, d'un intervalle ou d'une chaîne de caractères `x`
`x[i]`	string, list, range	donne le i^e élément d'une liste, d'un intervalle ou d'une chaîne de caractères `x`
`x.a`	struct	donne le champ `a` de la structure `x`

— **Routines d'entrées sorties sur le port série (UART0 de l'`ATmega328P`)**

Mots clef	Usage
`peek`	reçoit le dernier caractère en entrée du port série sans le supprimer du tampon d'entrée
`flush`	vide le tampon d'entrée du port série
`getc`	reçoit le dernier caractère en entrée sur le port série
`putc`	envoie un caractère sur le port série
`puts`	envoie une chaine de caractères sur le port série
`available`	donne le nombre d'octets (caractères) disponibles en entrée sur le port série

— **Routines d'entrées sorties sur les broches de l'Arduino (*pins*)**

Mots clef	Usage
`pinMode`	fixe l'état d'une broche en entrée (avec ou sans résistance de tirage) ou en sortie
`INPUT`	entrée
`INPUTPULLUP`	entrée avec une résistance de tirage
`OUTPUT`	sortie
`digitalWrite`	fixe une broche à un état haut ou bas
`digitalRead`	lit l'état d'une broche
`LOW`	état bas
`HIGH`	état haut
`analogReference`	configure la tension de référence utilisée pour une entrée analogique
`analogWrite`	envoie une tension sur une broche
`analogRead`	lit la tension envoyée à une broche et la convertit sur 10bits

— **Routines de délai**

Nous n'avons pas utilisé les interruptions ni les compteurs du micro-contrôleur, mais de simples boucles en comptant les cycles horloge du processeur. Ces fonctions ont une faible précision.

Mots clef	Usage
`delay_s`	boucle pendant plusieurs secondes
`delay_ms`	boucle pendant plusieurs milli-secondes

4 Annexe : Simulateur d'Arduino Uno

En absence d'une carte Arduino Uno, vous pouvez vous servir d'un simulateur. Dans ce projet, nous vous proposons le logiciel à code source ouvert *SimulIDE* permettant de simuler divers circuits électroniques et microcontrolleurs tels que ATmega328p de l'Arduino Uno.

4.1 Téléchargement et premier lancement

Des exécutables pré-compilés de *SimulIDE* sont disponibles pour les trois principaux systèmes d'exploitation, c'est-à-dire GNU Linux, Microsoft(R) Windows(R) et MacOS(R). Aucune procédure d'installation n'est nécessaire. La procédure de téléchargement ainsi que le premier lancement de *SimulIDE* sont décrits pour chaque système d'exploitation dans les sections qui suivent.

4.1.1 GNU Linux

Télécgarger le fichier simulide_0.4.15-SR9.AppImage[2]. Ensuite, rendez le fichier exécutable en effectuant la commande suivante depuis un terminal ouvert dans le dossier où se trouve le fichier téléchargé :

```
chmod +x simulide_0.4.15-SR9.AppImage
```

Finalement, pour lancer *SimulIDE*, il vous suffira de double-cliquer sur le fichier simulide_0.4.15--SR9.AppImage depuis votre explorateur de fichiers.

4.1.2 Microsoft(R) Windows(R) (64-bit)

Télécharger le fichier SimulIDE_0.4.15-SR9_Win64.zip[3]. Extrayez l'archive et double-cliquez sur le fichier bin\simulide.exe pour lancer *SimulIDE*.

4.1.3 MacOS(R)

Télécharger le fichier SimulIDE_0.4.15-SR9_MacOs.zip[4]. Extrayez l'archive et placer le dossier simulide.app dans le dossier Applications en effectuant la commande suivante à partir d'un terminal ouvert dans le dossier où l'archive a été extraite :

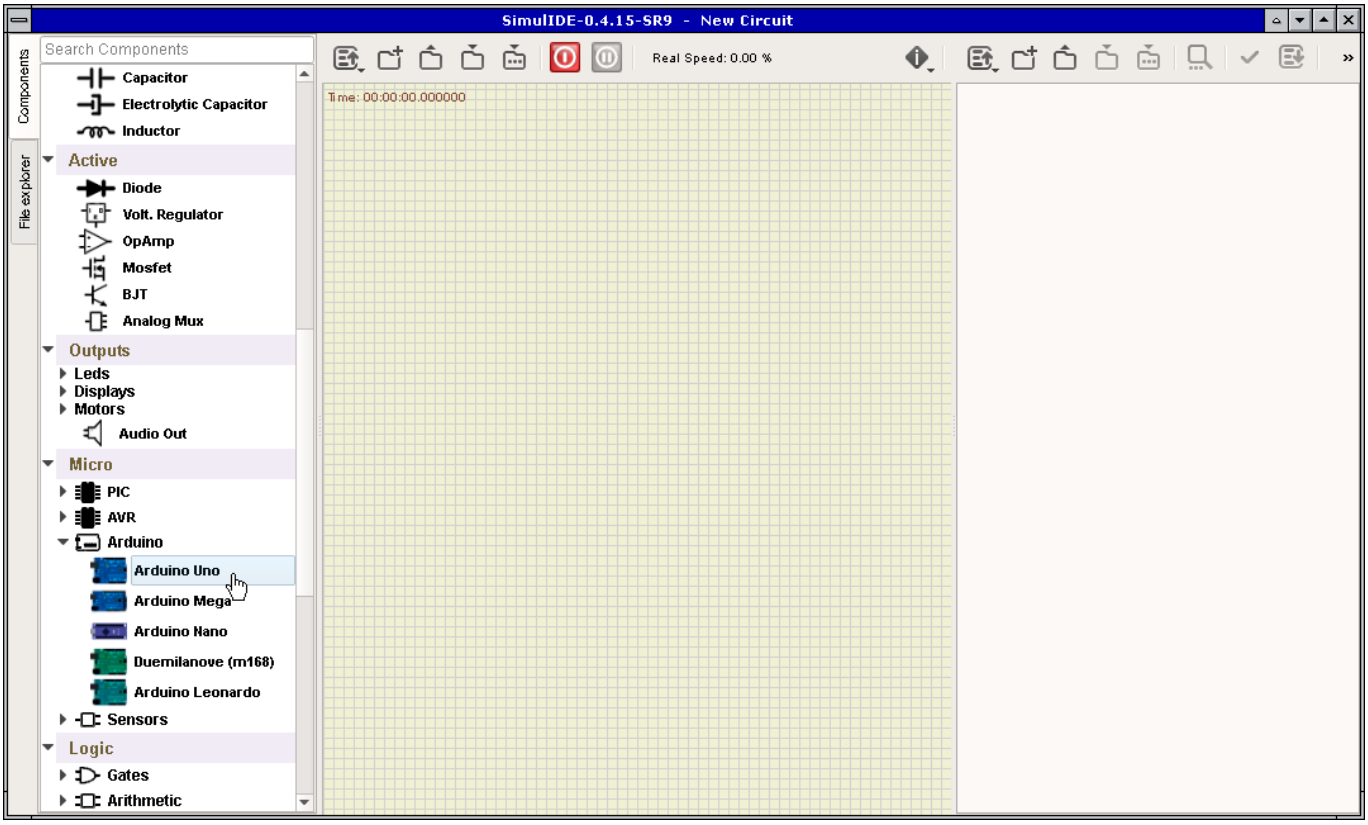
```
sudo cp -rf simulide.app /Applications
```

Pour ouvrir *SimulIDE* sur MacOS(R), naviguez dans le dossier Applications (voir la barre latérale gauche dans *Finder*). Puis, faite un clique droit sur l'application *SimulIDE* et cliquez sur « Ouvrir ». Une fenêtre d'avertissement apparaîtra. À ce moment-là cliquez sur le bouton « Ouvrir » et attendez que l'application se lance.

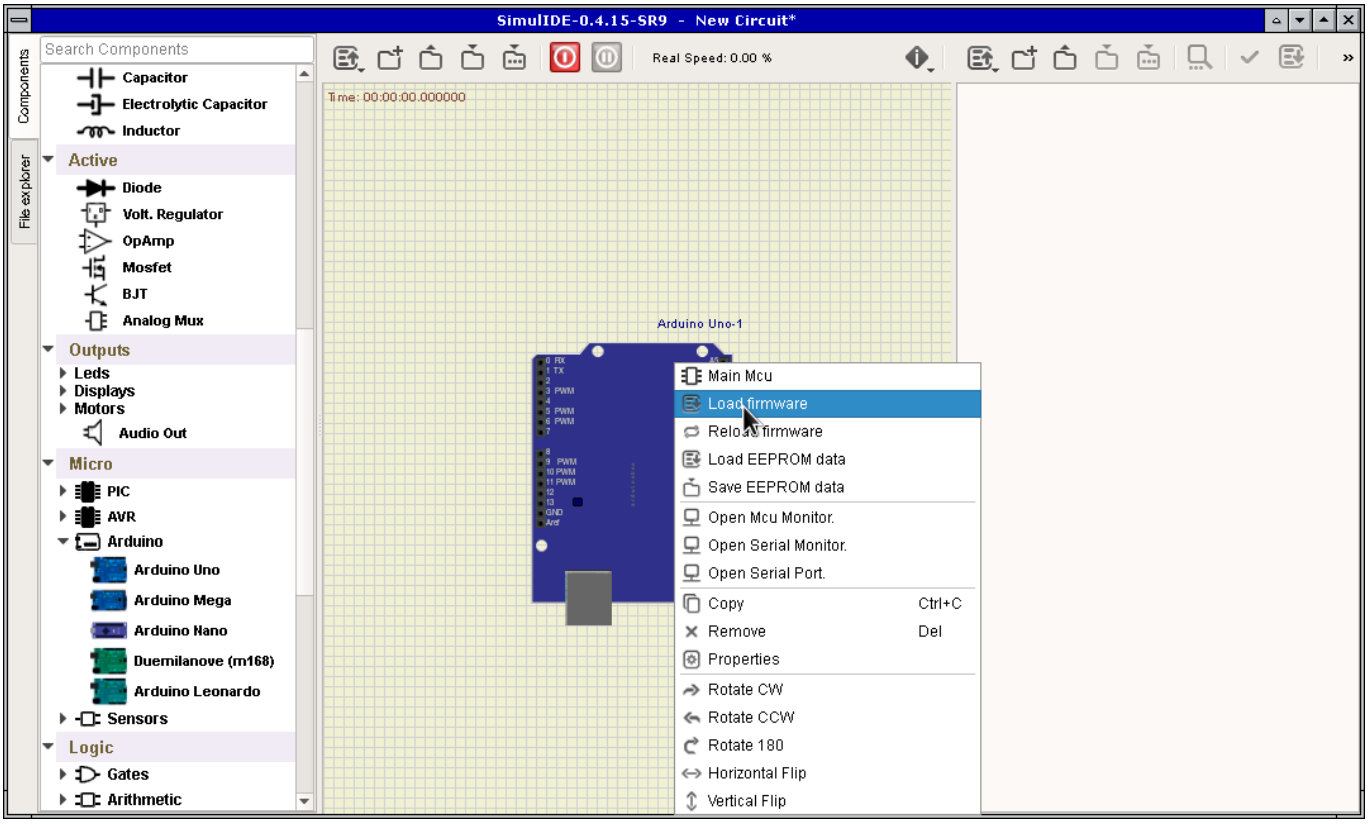
4.2 Utilisation

En ouvrant *SimulIDE* vous créez une nouvelle simulation de circuit. Depuis le menu latéral gauche, ajouter un Arduino Uno sur la grille au milieu de la fenêtre.

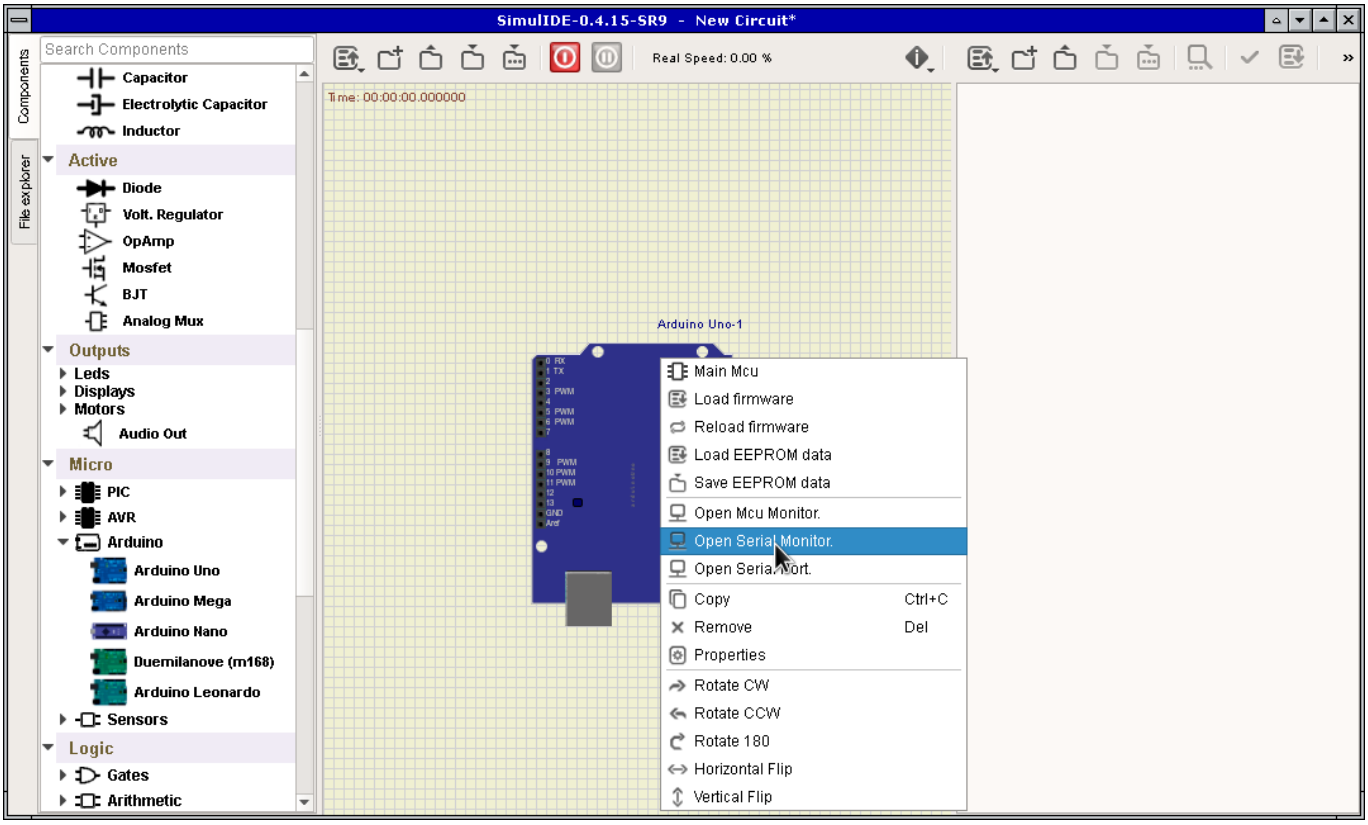
2. https://launchpad.net/simulide/0.4.15/0.4.15-stable/+download/simulide_0.4.15-SR9.AppImage
3. https://launchpad.net/simulide/0.4.15/0.4.15-stable/+download/SimulIDE_0.4.15-SR9_Win64.zip
4. https://launchpad.net/simulide/0.4.15/0.4.15-stable/+download/SimulIDE_0.4.15-SR9_MacOs.zip



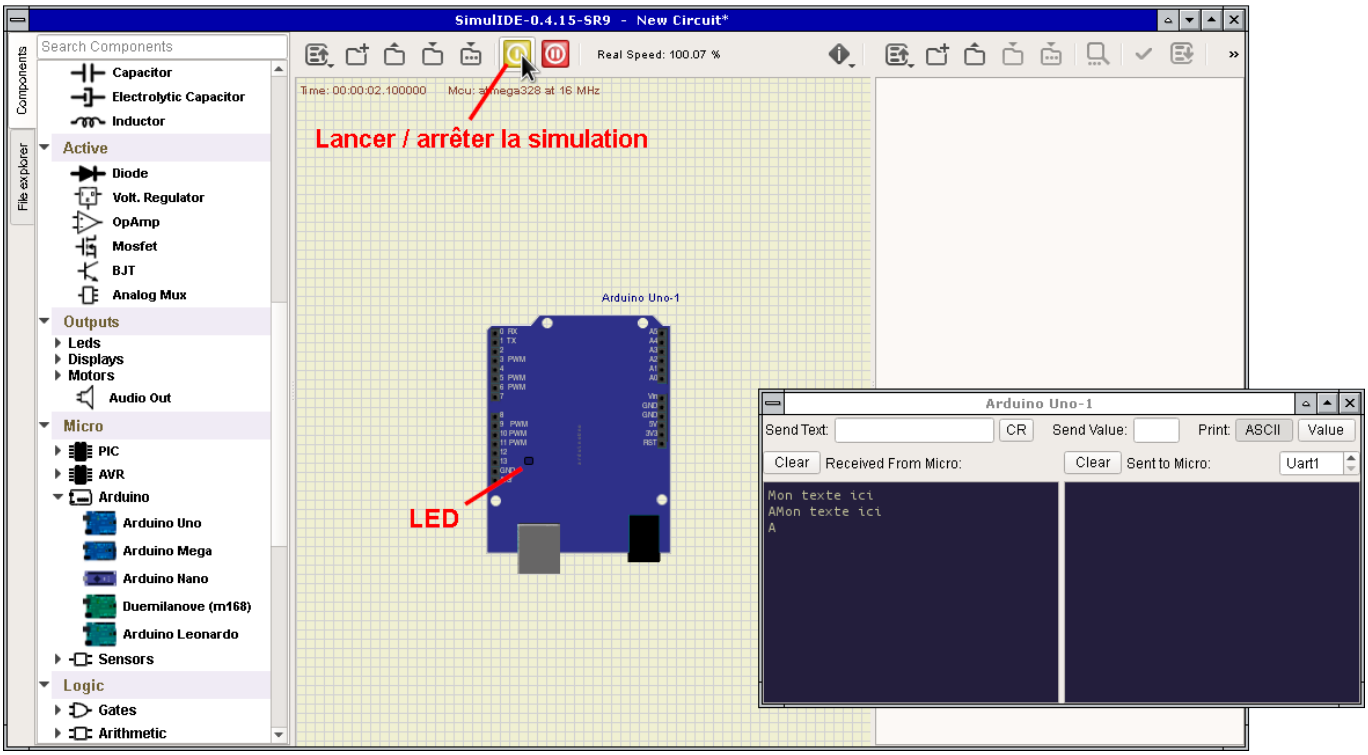
Pour télécharger un binaire au format HEX sur la carte Arduino Uno, cliqez droit sur cette dernière puis dans le menu contextuel qui appraît cliquez sur « Load firmware » et choisissez le fichier HEX souhaité à partir des fichiers sur votre ordinateur.



Afin de visualiser la sortie de la liaison en série du microcontrolleur, cliquez droit sur la carte Arduino Uno posée sur la grille puis dans le menu contextuel qui apparaît cliqez sur « Open Serial Monitor. ». Lors de l'exécution de la simulation, la sortie de la liaison en série s'affichera dans la fenêtre que vous venez d'ouvrir.



Pour lancer ou arrêter la simulation, cliquez sur le bouton prévu à cet effet comme le montre la figure ci-dessous.

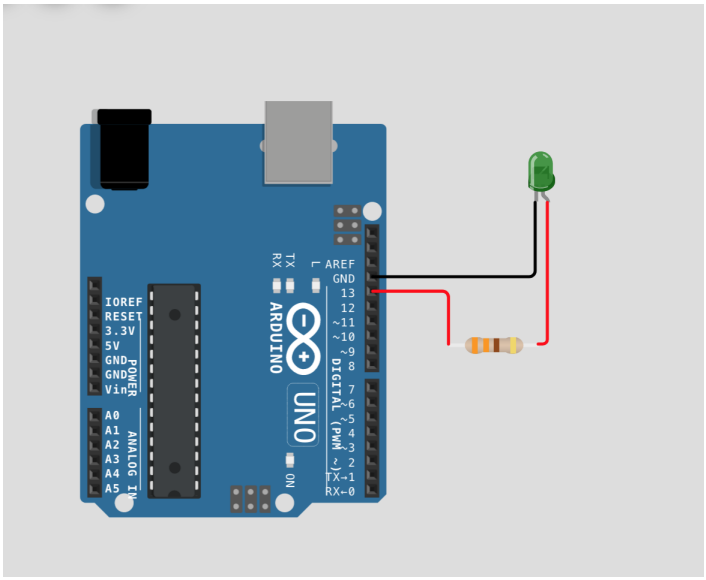


Enfin, il est possible de sauvegarder la simulation de circuit pour un usage ultérieur en utilisant les boutons au dessus de la grille.

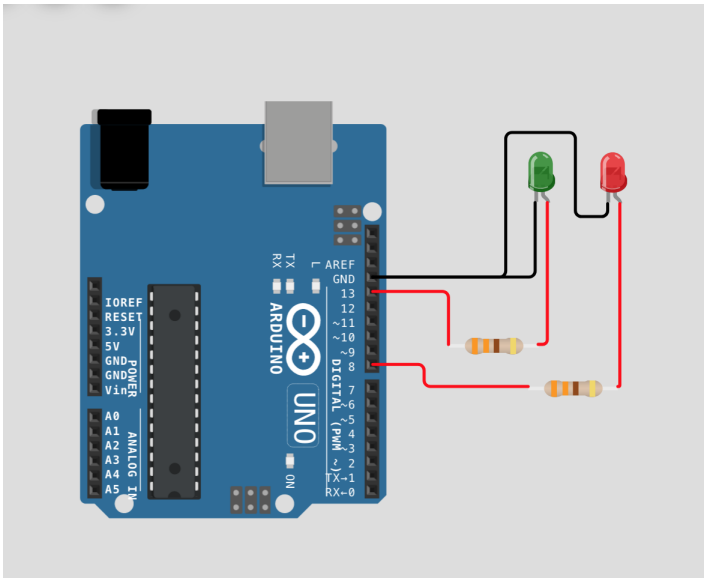
5 Annexe : Les cablages des quelques exemples fournis

1. `test-1.arduinoCode`

   On branche une LED sur la broche n°13. Une LED verte (resp. rouge) fonctionne optimalement sous $2.45V$ et $2.55mA$ (resp. $1.89V$ et $3.11mA$). Un vieux souvenir du lycée nous dit $U = RI$ et le microcontroleur fournit une sortie de 5 volts. Disons que nous voulons descendre de $3V$ pour avoir une différence de potentiel de $2V$ et avoir $3mA$. Ceci nous donne une résistance à placer en série de $1k\Omega$.

2. `test-2.arduinoCode`

   On branche une LED sur la broche n°13 et une autre sur la broche n°8.

3. `test-7.arduinoCode`

   Un bouton poussoir est branché de sorte qu'il ferme un circuit reliant la broche n°2 à la masse. La broche n°2 est reliée à une résistance de tirage en interne de l'Arduino Uno, c'est-à-dire une résistance placée entre $V_{cc} = 5V$ et la broche. Le circuit normalement ouvert, la broche reçoit un signal haut, circuit fermé la broche reçoit un signal bas $(0V)$.

   Ce montage avec `INPUTPULLUP` est à préférer d'un `INPUT` simple car sans résistance de tirage, la broche a une tension indéterminée lorsque le circuit est ouvert.