# Projet de Compilation

Lucas David, Sullivan Honnet, Théo Legras & Jules Vittone

## Part I

# Analyse lexicale

Premièrement, nous pouvons définir des expressions régulières intermédiaires qui nous seront utiles pour éviter les expressions trop complexes:

`{digit}` = `[0-9]`
`{letter}` = `[A-Za-z0-9_]`
`{newline}` = `[(\r\n?)|\n]`

Dans un second temps, nous ne traitons pas le commentaire comme un token mais nous le reconnaissons pour pouvoir l'éliminer de l'analyse:

`{commentary}` = `\/{2}[^{newline}]*{newline}|\/\*([^\*]*|\*[^\/])*\*\/`

Et finalement, pour définir comment s'organise le projet de compilation, il faut réfléchir aux tokens que nous allons reconnaître. Aussi, pour chacun des tokens reconnu le nombre de caractère devra être compté pour garder un contexte d'erreur lors de la compilation. En voici, un exemple d'approche:

Pour cela, nous indiquons l'option `--yylineno` lors de l'appel de `flex`, qui indiqué à flex d'incrémenter la valeur de `yylineno` à chaque saut de ligne.

Finalement, voici le tableau des différents tokens reconnus ainsi que leurs expressions régulières et types retournés respectifs (à noter que ∅ signifie que le token ne retourne pas de valeur):

| Token | Expression régulière | Type du token |
|---|---|---|
| Mots clés réservés | | |
| **CLASS** | `class` | ∅ |
| **DEF** | `def` | ∅ |
| **ELSE** | `else` | ∅ |
| **EXTENDS** | `extends` | ∅ |
| **IS** | `is` | ∅ |
| **IF** | `if` | ∅ |
| **OBJECT** | `object` | ∅ |
| **OVERRIDE** | `override` | ∅ |
| **VAR** | `var` | ∅ |
| **RETURN** | `return` | ∅ |
| **THEN** | `then` | ∅ |
| Opérateurs arithmétiques | | |
| '+' | \+ | ∅ |
| '-' | - | ∅ |
| '*' | * | ∅ |
| '/' | \/ | ∅ |
| Opérateurs relationnels | | |
| **RELATIONAL_OPERATOR** | <\|<=\|>=\|=\|<> | int |
| Autres opérateurs | | |
| **ASSIGNMENT** | := | ∅ |
| '.' | \. | ∅ |
| Idenifiants & constantes | | |
| **IDENTIFIER** | {letter}({letter}\|{digit})* | class std::string |
| **STRING** | "[^"]*" | class std::string |
| **CHAR** | '[^']+' | char |
| **INTEGER** | {digit}+ | int |
| **DOUBLE** | {integer}\.{digit}* | double |

On peut en déduire l'enum suivante (extrait du fichier `yytokentype.hpp`), qui servira dans le fonctionnement de `bison`:

```
enum yytokentype {
    ASSIGNMENT = 258, CLASS = 259,
    DEF = 260, ELSE = 261,
    EXTENDS = 262, IDENTIFIER = 263,
    IF = 264, INTEGER = 265,
    IS = 266, NEW = 267,
    OBJECT = 268, OVERRIDE = 269,
    RELATIONAL_OPERATOR = 270, RETURN = 271,
    STRING = 272, THEN = 273,
    THIS = 274, TYPENAME = 275,
    VAR = 276, unary = 277
};
```

# Part II

# Analyse syntaxique

## 1 Grammaire

### 1.1 Programme

Program := LOptDecls Block $

LOptDecls := LDecls | $\varepsilon$
LDecls := Decl LDecls | Decl
Decl := Class | Object

### 1.2 Déclarations

#### 1.2.1 Déclaration d'une classe

Class := **CLASS TYPENAME** '(' LOptParamDecl ')' OptExtends **IS** '{' LOptField ClassConstructor LOptMethod '}'
OptExtends := **EXTENDS TYPENAME** | $\varepsilon$

LOptParamDecl := LParamDecl | $\varepsilon$
LParamDecl := ParamDecl , LParamDecl | ParamDecl
ParamDecl := OptVar **IDENTIFIER** : **TYPENAME**
OptVar := **VAR** | $\varepsilon$

ClassConstructor := **DEF TYPENAME** '(' LOptParamDecl ')' ':' **TYPENAME** '(' LOptExpr ')' **IS Block** | **DEF TYPENAME** '(' LOptParamDecl ')' **IS** Block

#### 1.2.2 Déclaration d'un objet

Object := **OBJECT TYPENAME IS** '{' LOptField ObjectConstructor LOptMethod }

ObjectConstruct := **DEF IDENTIFIER IS** '{' Bloc '}'

#### 1.2.3 Déclaration d'un champ

LOptField := LField | $\varepsilon$
LField := Field LField | Field
Field := **VAR IDENTIFIER** ':' **TYPENAME** ';'

#### 1.2.4 Déclaration d'une méthode

LOptMethod := LMethod | $\varepsilon$
LMethod := Method LMethod | Method
Method := OptOverride **IDENTIFIER** '(' LOptParamDecl ')' ':' **IDENTIFIER ASSIGNMENT** Expr
  | OptOverride **IDENTIFIER** '(' LOptParamDecl ')' OptReturn **IS** Bloc
OptOverride := **OVERRIDE** | $\varepsilon$
OptReturn := ':' **IDENTIFIER** | $\varepsilon$

## 1.3 Expressions et instructions

### 1.3.1 Expressions

On connait le principe pour faire apparaitre la priorité et l'associativité des opérateurs mais vu que Bison l'intègre "en dehors" de la définition des règles on va faire de même (pour consulter notre liste la priorité et l'associativité des opérateurs, autant directement consulter celle du C++: `https://en.cppreference.com/w/cpp/language/operator_precedence`).

Expr := Expr **RELATIONAL_OPERATOR** Expr
| Expr '+' Expr
| Expr '-' Expr
| Expr '*' Expr
| Expr '/' Expr
| **NEW IDENTIFIER** '(' LOptParam ')'
| '+' Expr
| '-' Expr
| '(' **IDENTIFIER** Expr ')'
| **IDENTIFIER** ':' **IDENTIFIER**
| **IDENTIFIER** ':' **IDENTIFIER** '(' LOptParam ')'
| '(' Expr ')'
| **IDENTIFIER**
| **INTEGER**
| **STRING**

### 1.3.2 Instructions

Instr := Expr ';'
| Bloc
| **RETURN** ';'
| Expr **ASSIGNMENT** Expr ';'
| **IF** Expr **THEN** Instr **ELSE** Instr

Bloc := '{' LOptInst '}' | '{' LOptVarDecl **IS** LInst '}'

LOptVarDecl := LVarDecl | $\varepsilon$
LVarDecl := VarDecl LVarDecl | VarDecl
VarDecl := **IDENTIFIER** ':' **IDENTIFIER** Expr ';'
| **IDENTIFIER** ':' **IDENTIFIER** ';'