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]`
`{upperletter}` = `[A-Z]`
`{lowerletter}` = `[a-z]`
`{letter}` = `{lowerletter}|{upperletter}`
`{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		
TYPENAME	{upperletter}({letter}|{digit})*	class std::string
IDENTIFIER	{lowerletter}({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 | ε
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** | ε

LOptParamDecl := LParamDecl | ε
LParamDecl := ParamDecl , LParamDecl | ParamDecl
ParamDecl := OptVar **IDENTIFIER** : **TYPENAME**
OptVar := **VAR** | ε

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 TYPENAME IS** '{' Bloc '}'

1.2.3 Déclaration d'un champ

LOptField := LField | ε
LField := Field LField | Field
Field := **VAR IDENTIFIER** ':' **TYPENAME** ';'

1.2.4 Déclaration d'une méthode

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

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++ qui est un modèle fiable): `https://en.cppreference.com/w/cpp/language/operator_precedence`).

LOptExpr := LExpr | ε
LExpr := Expr ',' LExpr | Expr
Expr := Expr **RELATIONAL_OPERATOR** Expr
| Expr '+' Expr
| Expr '-' Expr
| Expr '*' Expr
| Expr '/' Expr
| **NEW TYPENAME** '(' LOptParam ')'
| '+' Expr
| '-' Expr
| '(' **TYPENAME** Expr ')'
| Expr '.' **IDENTIFIER**
| **TYPENAME** '.' **IDENTIFIER**
| Expr '.' **IDENTIFIER** '(' LOptParam ')'
| **TYPENAME** '.' IDENTIFIER '(' LOptExpr ')'
| '(' Expr ')'
| **STRING**
| **INTEGER**
| **IDENTIFIER**
| **THIS**

1.3.2 Instructions

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

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

LOptVarDecl := LVarDecl | ε
LVarDecl := VarDecl LVarDecl | VarDecl
VarDecl := **IDENTIFIER** ':' **IDENTIFIER** Expr ';'
| **IDENTIFIER** ':' **IDENTIFIER** ';'

On peut tenir le tableau suivant qui indique pour chaque non-terminal à quel type de structure, il correspond:

Structure	Non-terminaux
bool	OptVar OptOverride
std::string*	OptExtends OptReturn
std::vector<Parameter>*	LOptParamDecl LParamDecl
Parameter*	ParamDecl
std::vector<Variable>*	LOptVar LVar LOptField LField
Variable*	Var Field
std::vector<Type*>*	LOptDecls LDecls
Type*	Decl
Class*	Class
Object*	Object
std::vector<Method>*	LOptMethod LMethod
Method*	Method ClassConstructor ObjectConstructor
std::vector<Tree*>*	LOptInst LInst LOptExpr LExpr
Tree*	Inst Expr
Block*	Block

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

```
typedef union {
    bool Boolean;   /* Valeur booléenne. */
    char Char;    /* Caractère isolé. */
    int Integer;   /* Valeur entière. */
    std::string *String; /* Chaîne de caractère. */

    std::vector<Parameter> *PParamList;
    Parameter *PParam;
    std::vector<Variable> *PVarList;
    Variable *PVar;

    std::vector<Type*> *PTypeList;
    Type *PType;
    Class *PClass;
    Object *PObject;

    std::vector<Method> *MethodList;
    Method *PMethod;

    std::vector<Tree*> *PTreeList;
    Tree *PTree;
    Block *PBlock;
} YYSTYPE;
```

De plus, chacune des expressions à un code d'opération défini par une enum pour qualifier son comportement dans la structure Tree (extrait du fichier `optype.hpp`)

```
enum optype {
    /* CLASSES & OBJECTS */
    instanciation, cast, member_access, static_member_access, method_call, static_method
    /* ASSIGNMENT */
    assignment,
    /* ARITHMETIC */
    unary_plus, unary_minus,
    multiplication, division,
    addition, substraction,
    /* RELATIONAL_OPERATOR */
    less_strict, less_equal, greater_strict, greater_equal,
    equal, not_equal,
    /* MISCELLANEOUS */
    if_then_else, return_call, inst_block,
    /* CONSTS & IDENTIFIER */
    integer, string, identifier
};
```

Part III

Conclusion

1 Choix d'implémentation

Pour le projet, on a fait le choix de faire le compilateur en C++ pour nous permettre d'utiliser l'héritage entre les structures. Dans le projet, nous avons séparé les structures en différents morceaux, nous avons l'Arbre de Syntaxe Abstraite contenant toutes les variables et les opérations réalisées. Les structures 'Class' et les 'Object' ont une base 'Type' pour nous permettre de traiter certains comportements par défaut. On peut citer, la gestion des champs et des méthodes, mais aussi les vérifications contextuelles ou la génération de code.

La classe 'Method' nous permet de définir ce qui se rapporte à ces dernières. Elle va nous permettre de généraliser des comportements. La structure 'Constructeur' hérite de méthode (car c'est une méthode avec un comportement particulier), qui elle même est dérivée en 'ClassConstructor' et 'ObjectConstructor', afin de pouvoir modéliser les spécificités de chacune de ces entités.

La structure 'Environment' va nous servir à passer les informations qui nous serviront lors des vérifications contextuelles : les types connus et les variables qui sont actuellement définies. À noter que les variables sont représentées sous forme de vecteur afin de pouvoir gérer le masquage : les variables sont ajoutées à la fin et supprimées à la fin du bloc (à la manière d'une pile) qui définit leur portée. Pour accéder à une variable, on parcourt ce vecteur en partant de la fin pour tomber en premier sur la dernière variable définie (qui donc masque les autres).

Les structures 'Bloc' et 'Tree' sont des structures de stockage. 'Tree' va nous permettre de stocker les différentes instructions de notre programme. 'Bloc' est un ensemble de déclarations optionnelles suivies d'une liste d'instructions optionnelles. Cette dernière structure nous sert essentiellement pour les problèmes de portée dans l'environnement.

2 Avancement : ça compile, mais ça ne compile pas

En ce qui concerne l'avancement, nous avons fini les parties lexicales et syntaxiques. Ces parties ont été testées et sont fonctionnelles.

La partie génération de code a été faite pour les arbres, et il reste à le faire pour les différents types, classes et pour les variables. La génération de code fonctionne pour les instructions conditionnelles et pour les calculs arithmétiques. Les structures nécessaires à la gestion des variables existent, mais ne sont pas utilisées.

À propos des vérifications contextuelles : les vérifications n'ont pas été totalement faites ni testées.

Néanmoins, nous avons fini de développer un grand nombre de vérifications. Entre autres : nous avons fini la vérification pour la portée des variables (bloc par bloc). La vérification que les arbres sont corrects : quand on appelle une méthode elle appartient bien à la classe qui l'appelle ou à une des super classes de cette dernière. Nous n'avons pas la vérification qui permet de passer en paramètre une classe dérivée de la classe appelée demandée par la signature (il faut la classe exacte). On peut vérifier la cohérence des types lors de l'appel ou de l'initialisation de ces derniers (donc dans la classe Type directement). Pour la déclaration des types : la correcte déclaration est vérifiée correctement, on vérifie qu'il n'y a pas d'héritage circulaire, que le type n'existe pas déjà, que si une classe dérive d'une autre classe cette dernière existe et qu'elle soit héritable (que ce ne soit pas un Object). Puis chaque type va vérifier la validité de ses méthodes en appelant la fonction correspondante. Pour une méthode, on vérifie les types et la portée des variables utilisées ou passées en paramètres. De manière générale, chaque élément vérifie de manière récursive qu'il est correct. On a beaucoup d'affichage pour donner la ligne et le type d'erreur.

3 Résumé des tâches

Participation de Lucas : Lucas a fait l'analyse lexicale et a corrigé l'analyse syntaxique et la génération de code. Il a également réécrit le rapport au format LaTeX.

Participation de Théo : Théo s'est occupé des vérifications contextuelles, du support et du débogage.

Participation de Jules : Jules a écrit la première version de l'analyse syntaxique et a fait les tests des différentes versions de l'analyse syntaxique ainsi que la génération de code.

Participation de Sullivan : Sullivan a participé à l'analyse syntaxique et a écrit la liste des vérifications contextuelles à faire avec Théo.