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:

Soit $numLine$, $numCharOfLine$, $numChar$ des entiers initialisés à 0.
Chaque fois que `{newline}` est reconnu $numLine$ est incrémenté de 1.
Chaque fois que `{commentary}` est reconnu $numLine$ est incrémenté de 1 ou plus selon si le commentaire est multilignes ou non.
Pour chaque token reconnut $numCharOfLine$ & $numChar$ sont incrémenté du nombre de caractères du token (`strlen(yytext)`).
Et finalement pour tout autre caractère simple reconnu $numCharOfLine$ & $numChar$ sont incrémentés de 1.

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		
PLUS	`\+`	∅
MINUS	`-`	∅
MULTIPLY	`\*`	∅
DIVIDE	`\/`	∅
Opérateurs relationnels		
RELATIONAL_OPERATOR	`=|<>`	enum operation::relational
Autres opérateurs		
ASSIGNMENT	`:=`	∅
MEMBER_ACCESS	`\.`	∅
Idenifiants & constantes		
IDENTIFIER	`{letter}({letter}|{digit})*`	class std::string
STRING	`"[^"]*"`	class std::string
CHAR	`'[^']+'`	char
INTEGER	`{digit}+`	int
DOUBLE	`{integer}\.{digit}*`	double

Part II

Analyse syntaxique

1 Déclarations

1.1 Préambule

```
LOptParam  := LParam
            | ε
LParam     := Param , LParam
            | Param
Param      := IDENTIFIER
```

```
LOptParamDecl := LParamDecl
               | ε
LParamDecl    := ParamDecl , LParamDecl
               | ParamDecl
ParamDecl     := OptVar IDENTIFIER : IDENTIFIER
OptVar        := VAR
               | ε
```

1.2 Déclaration d'une classe

```
Class        := CLASS IDENTIFIER ( LOptParamDecl ) OptExtends IS { ClassDef }
OptExtends   := EXTENDS IDENTIFIER
              | ε
ClassDef     := LOptField ClassConstructor LOptMethod
ClassConstruct := DEF IDENTIFIER ( LOptParamDecl ) OptSuper IS { Bloc }
OptSuper     := IDENTIFIER ( LOptParam )
```

1.3 Déclaration d'un objet

```
Object       := CLASS IDENTIFIER IS { ObjectDef }
OptExtends   := EXTENDS IDENTIFIER
              | ε
ObjectDef    := LOptField ObjectConstructor LOptMethod
ObjectConstruct := DEF IDENTIFIER IS { Bloc }
```

1.4 Déclaration d'un champ

```
LOptField  := LField
            | ε
LField     := Field LField
            | Field
Field      := VAR IDENTIFIER : IDENTIFIER ;
```

1.5 Déclaration d'une méthode

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

1.6 Expressions et instructions

1.6.1 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 **PLUS** Expr
| Expr **MINUS** Expr
| Expr **MULTIPLY** Expr
| Expr **DIVIDE** Expr
| **NEW IDENTIFIER** (LOptParam)
| **PLUS** Expr
| **MINUS** Expr
| (**IDENTIFIER** Expr)
| **IDENTIFIER MEMBER_ACCESS IDENTIFIER**
| **IDENTIFIER MEMBER_ACCESS IDENTIFIER** (LOptParam)
| (Expr)
| **IDENTIFIER**
| **INTEGER**
| **STRING**

1.6.2 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** ;