Projet de compilation

Benjamin Maurice
Thomas Paulet
David Rubino
Mickaël Thomas

12 mai 2014

Introduction

L'objectif de ce projet est la réalisation d'un compilateur du langage LEAC (Langage Elémentaire Algorithmique pour la Compilation) vers du code assembleur microPIUP.

Dans ce cadre, nous avons dans un premier temps utilisé l'outil ANTLR dans sa version 4 pour générer un analyseur syntaxique puis dans un second temps, nous nous sommes appuyés sur le langage Java pour l'analyse sémantique, la construction de la table des symboles, les erreurs sémantiques et la génération du code machine.

En ce qui concerne la gestion de projet, nous avons utilisé le gestionnaire de version Git sur Bitbucket[1] et avons copié notre dépôt sur la forge de TELECOM Nancy.

Notre compilateur implémente l'ensemble des fonctionnalités du langage : fonctions, tableaux, expressions, passage par référence.

Nous allons donc présenter notre réalisation en suivant ce plan :

- construction de la grammaire du langage
- la structure de l'arbre abstrait et de la table des symboles définis à notre convenance
- les erreurs traitées par notre compilateur
- les schémas de traduction les plus pertinents
- quelques jeux d'essais et limites éventuelles
- une évaluation de la répartition du travail

[1] http://bitbucket.org/

Construction de la grammaire

Particularités d'ANTLR4

Nous avons décidé de construire notre grammaire en utilisant ANTLR 4, qui accepte les grammaires directement récursives gauches et permet une résolution des conflits basée sur l'ordre d'apparition des règles. Cela permet de grandement simplifier le travail puisqu'il n'y a pas besoin de dérouler la grammaire. Par exemple, on peut écrire :

```
expr : expr ('*' | '/') expr
     | expr ('+' | '-') expr
     ;
```

ANTLR reconnait alors qu'il s'agit d'un opérateur binaire, et transforme la grammaire en utilisant des prédicats sémantiques qui permettent de désactiver des alternatives au fur et à mesure que le niveau d'imbrication augmente.

Dans l'extrait suivant, les suffixes comme **# blockInstr** permettent de nommer une alternative afin de pouvoir la distinguer lors de l'analyse de l'arbre syntaxique. De même on peut nommer les enfants d'une règle de façon à les distinguer (instrIf=instr et instrElse=instr).

```
compInstr : '{' sequence? '}' # blockInstr
          | 'while' expr 'do' instr # whileInstr
          | 'if' expr 'then' instrIf=instr ('else' instrElse=instr)? # ifInstr
          ;
```

Modifications apportés au langage Leac

Nous avons souhaité apporter quelques changements au langage ainsi qu'à sa grammaire :

- Gestion des chaînes de caractère littérales (mais pas de possibilité de les modifier)
- Ajout de l'instruction "writeln" qui fonctionne comme "write" mais qui ajoute un saut de ligne
- Possibilité d'utiliser write sur une expression (au lieu de uniquement lvalue et constante)
- Autorisation de l'utilisation d'un ";" facultatif en fin de bloc.

Grammaire

```
grammar Leac;
program     : 'program' IDF varDeclList funDeclList instr EOF ;
varDeclList : varDecl* ;
varDecl     : 'var' identList ':' typeName ';' ;
identList   : IDF (',' IDF)* ;
typeName    : atomType | arrayType ;
```

```
atomType    : 'void' | 'bool' | 'int' | 'string' ;
arrayType   : 'array' '[' rangeList ']' 'of' atomType;
rangeList   : boundRange (',' boundRange)* ;
boundRange  : bound '..' bound ;
bound       : MINUS? INTCST ;
funDeclList : funDecl* ;
funDecl     : 'function' IDF '(' argList ')' ':' atomType varDeclList instr ;
argList     : ( arg (',' arg)* )? ;
arg         : IDF ':' typeName | REF IDF ':' typeName ;
compInstr   : '{' sequence? '}' # blockInstr
            | 'while' expr 'do' instr # whileInstr
            | 'if' expr 'then' instrIf=instr ('else' instrElse=instr)? # ifInstr ;
instr       : singleInstr (';'?) | compInstr ;
singleInstr : 'return' expr? # returnInstr
            | 'read' lvalue  # readInstr
            | 'write' expr # writeInstr
            | 'writeln' expr? # writeInstr
            | lvalue '=' expr # affectInstr
            | IDF '(' exprList? ')' # funCallInstr ;
sequence    : singleInstr (';' sequence)? (';'?)
            | compInstr sequence? ;
lvalue      : IDF ('[' exprList ']')? ;
exprList    : expr (',' expr)* ;
expr        : cste # cstExpr
            | '(' expr ')' # subExpr
            | IDF '(' exprList? ')' # funCallExpr
            | lvalue # lvalueExpr
            | IDF # idfExpr
            /* opérateurs ordonnés par précédance décroissante (spécifique ANTLR4) */
            |<assoc=right> expr op='^' expr # binaryExpr
            | op=('-' | 'not') expr     # unaryExpr
            | expr op=('*' | '/') expr  # binaryExpr
            | expr op=('+' | '-' ) expr # binaryExpr
            | expr op=('<' | '<=' | '>' | '>=') expr # binaryExpr
            | expr op=('==' | '!=') expr # binaryExpr
            | expr op='and' expr # binaryExpr
            | expr op='or' expr  # binaryExpr ;
cste        : INTCST | BOOLCST | STRCST ;

/* Règles lexicales */

STRCST   : '"' .*? '"';
INTCST   : [0-9]+;
BOOLCST  : 'true' | 'false' ;
IDF      : [a-zA-Z][a-zA-Z0-9]*;
COMMENT  : '/*' .*? '*/' -> skip;
WS       : [ \r\t\n]+ -> skip;
```

Structure de l'arbre abstrait et création de la TDS

La contrepartie de l'utilisation d'ANTLR4 plutôt qu'ANTLR3 est que ANTLR4 ne supporte pas la génération d'AST, il ne fournit qu'un arbre syntaxique que le programme peut parcourir en utilisant le pattern *visitor* ou un *listener*.

Pour le *listener*, on écrit une classe qui hérite de **X**Listener, où **X** désigne le nom de la grammaire, et qui comporte pour chaque alternative une méthode enter**X**(Context) et exit**X**(Context), où **X** désigne le nom de l'alternative (qui est par défaut le nom de la règle si on ne la nomme pas). Le contexte représente un nœud de l'arbre. ANTLR s'occupe alors de parcourir l'arbre syntaxique d'appeler les méthodes correspondant aux nœuds rencontrés. L'arbre est ainsi parcouru en totalité, même si on n'implémente pas toutes les méthodes (qui comportent une implémentation de base vide).

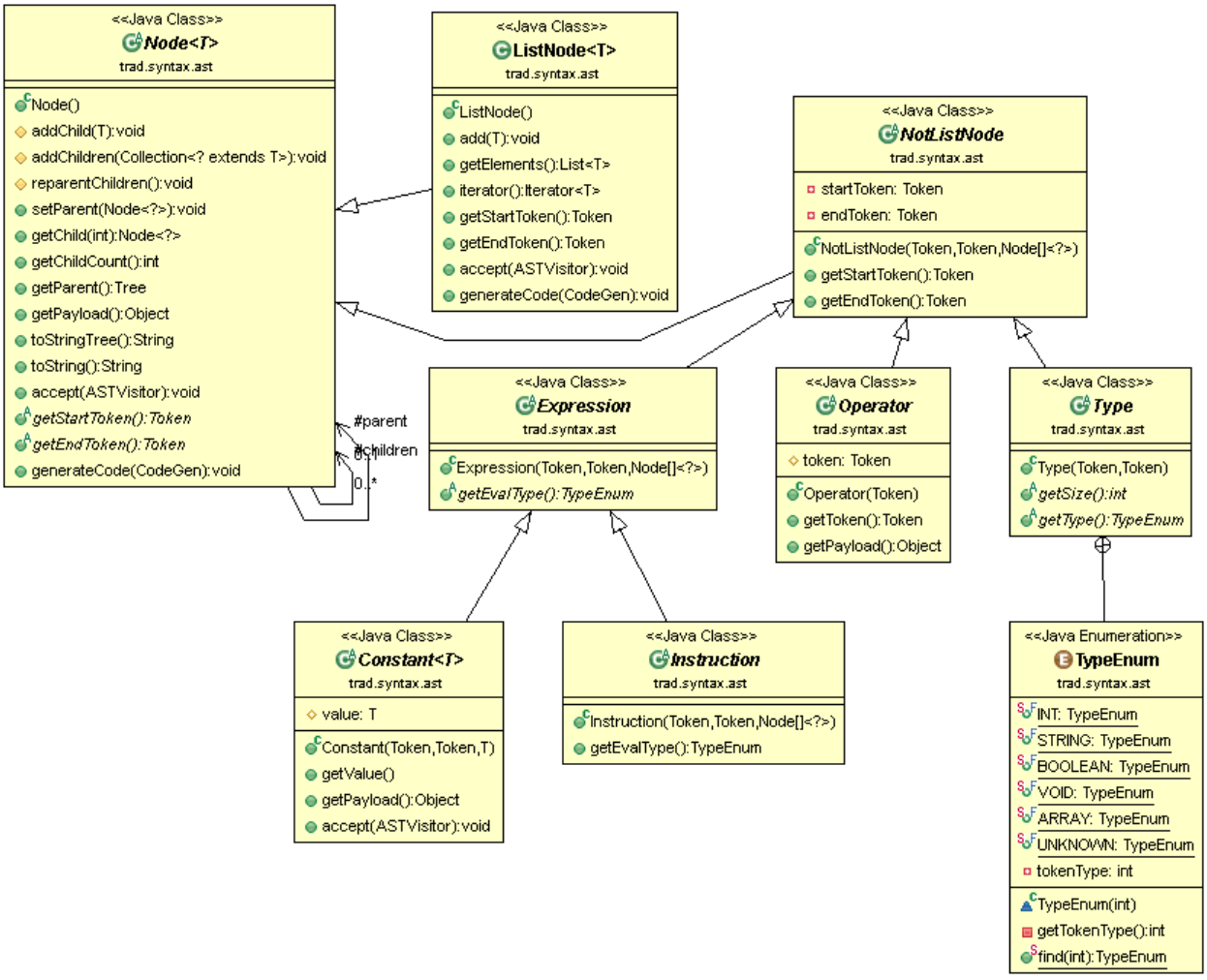
En ce qui concerne le *visitor*, on implémente une classe **X**BaseVisitor qui comporte des méthodes visit**X**() où **X** désigne le nom de l'alternative. Cette méthode est appelée par un nœud lorsque l'on visite celui-ci. Par exemple, *ProgramContext.accept(v)* où *v* est notre classe implémentant *BaseVisitor appelle *v.visitProgram(this)*. La méthode *visitProgram()* peut alors décider quel nœud de l'arbre elle souhaite visiter en appelant this.visit(c) qui est équivlant à c.accept(this). Par défaut, les méthodes visit**X**() ont une implémentation qui appelle visit() sur tous les enfants du nœud, ce qui permet de ne pas avoir à écrire une implémentation lorsque l'on est pas intéressé par des nœuds intermédiaires. Toutes ces méthodes renvoient un argument de type générique T (déterminé par notre implémentation de la classe **X**BaseVisitor), Cela nous permet donc d'associer à chaque nœud de l'arbre syntaxique, un nœud de l'arbre abstrait (dont on peut définir la structure). On peut donc décider dans chaque méthode visit**X**() de générer un nouveau nœud de l'AST, ou de renvoyer un nœud déjà existant, ainsi que de décider quels enfants on veut inclure dans ce nœud.

Exemple :

```
@Override
public Tree visitAffectInstr(AffectInstrContext ctx) {
    return new InstructionList(new AffectInstr(
            ctx.start, ctx.stop,
            visitLvalue(ctx.lvalue()),
            (Expression) visit(ctx.expr())));
}
```

Nous avons choisi d'utiliser le pattern *visitor* car il permet plus de flexibilité, et nous permet de générer sans effort un AST. On remarquera que cela constitue un équivalent assez proche de la génération d'AST d'ANTLR3 si ce n'est qu'il faut implémenter notre propre structure d'AST.

Les structures de bases définissant l'AST sont définies dans le package trad.syntax.ast. Leur structure est présenté dans l'UML ci-dessous.



Ces classes abstraites (à part ListNode<T>) définissent les différents types de structure que l'on trouve dans l'arbre abstrait et qui sont implémentées dans le package trad.syntax.ast.impl. Toute structure hérite de Node<T>, qui implémente la structure Tree définie par ANTLR. Un noeud spécifique peut avoir une collection d'enfants si cette collection est du type T (utilisation de la généricité). ListNode<T> (classe non abstraite) hérite de Node<T> et implémente Iterable. Elle définit la liste des noeuds d'un Node, qui sont des objets de type générique T. On peut ensuite invoquer les méthodes d'Iterable pour le parcours de la liste. NotListNode hérite aussi de Node, elle s'utilise pour les instructions utilisant des tokens de début et fin (if, for, function…). Toute structure hérite soit de ListNode, soit de NotListeNode.

La classe Type permet de définir tous les types de la grammaire (INT, STRING, BOOLEAN, VOID, ARRAY, UNKNOWN).

La classe Expression définit toutes les expressions (comme les fonctions) et possède une méthode pour retourner le type d'une expression. La classe Instruction hérite de Expression mais, contrairement a une Expression qui peut avoir tout type de retour, une Instruction a uniquement le type VOID comme type de retour. Les 2 objets ont la même structure: ils sont définis par un token de début et de fin et un ou plusieurs noeuds.
La classe Declaration hérite de NotListNode et permet de définir la déclaration d'une ou plusieurs variables.
La classe Operator permet de définir les différents opérateurs de la grammaire (+, *...) a l'aide d'un Token.
La classe Constant hérite de Expression et définit les constantes du programme.

Le package trad.syntax.ast.impl implémente les différentes instructions de la grammaire en utilisant les relations d'héritage avec les classes définies ci-dessus. Ces classes sont décrites dans le tableau synthétique ci-dessous. Dans chacune de ces classes sont implémentées les méthodes nécessaires a la génération de code (discutées dans la partie des schémas de traduction).

Classe	**Hérite de**	**Responsabilité**
AffectInstr	Instruction	Défini la relation d'affectation entre une LValue et une Expression
Argument	NotListNode	Défini l'argument d'une fonction. Un booléen permet de savoir s'il est référencé.
ArraySubscript	LValue	Permet de définir l'allocation mémoire pour un tableau
ArrayType	Type	Définit le type et les bornes d'un tableau
AtomType	Type	Définit le type d'un Atom en fonction de son Token. On peut donc retrouver s'il s'agit d'un int, bool ou string.
BinaryExpr	Expression	Définit les expressions binaires arithmétiques et booléens (ex: 2 + 3)
BinaryOperator	Operator	Définit les opérateurs binaires arithmétiques et booléens (+ * …)

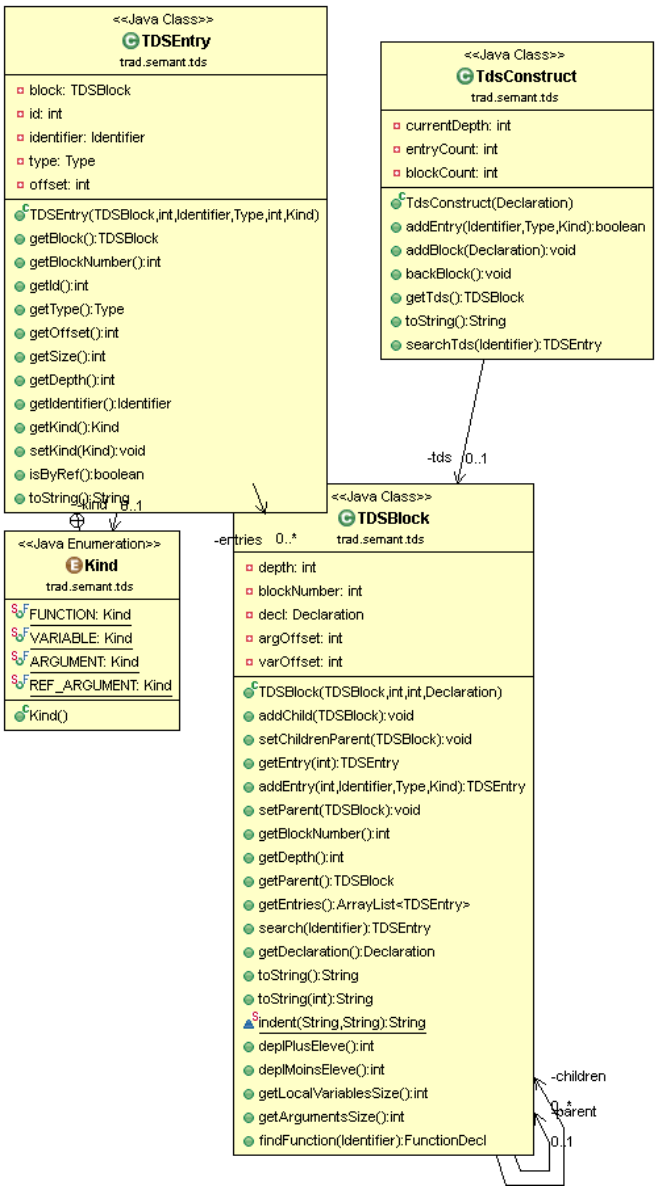
BoolConstant	Constant	Définit les constantes booléennes
BoundRange	NotListNode	Définit l'intervalle des valeurs que peut prendre un indice de tableau
FunctionCall	Instruction	Définit l'appel d'une fonction avec la liste des arguments que prend la fonction
FunctionDecl	Declaration	Définit la déclaration d'une fonction avec son type, son nom, sa liste de variables et d'arguments et ses instructions
Identifier	LValue	Définit le nom d'une variable, constante ou fonction
IfInstr	Instruction	Définit l'instruction if avec son paramètre et ses instructions
InstructionList	ListNode	Définit les instructions d'une fonction, d'un bloc ou d'une boucle. Chaque liste possède a la base une instruction de départ auxquelles on ajoute une liste de nouvelles instructions.
IntConstant	Constant	Définit les constantes entières
LValue	Expression	Classe abstraite pour représenter une valeur (int, string ou boolean)
Program	Declaration	Définit la structure d'un programme avec un nom, une liste d'instructions, de variables et de fonctions
ReadInstr	Instruction	Utilisée pour la lecture d'une valeur
ReturnInstr	Instruction	Définit l'instruction de retour d'un programme

StringConstant	Constant	Définit les constantes pour les chaînes de caractère
UnaryExpr	Expression	Définit les opérations unaires (ex: -3)
UnaryOperator	Operator	Définit les opérateurs unaires - et NOT
VariableDecl	NotListNode	Définit la déclaration d'une variable
VariableDeclList	ListNode	Définit la liste de déclaration des variables pour un progamme, bloc ou fonction
WhileInstr	Instruction	Définit l'instruction while avec son paramètre et ses instructions
WriteInstr	Instruction	Définit les instructions write et writeln

L'AST est construit a l'aide d'un visiteur définit dans la classe ASTConstructVisitor. Un Program est passé en paramètre dans la classe. La méthode makeListNode permet de construire tous les noeuds du programme. Puis, sur chacun de ces noeuds, en fonction de leur type, on appelle la fonction visit[Type] sur le noeud (ex: visitCste pour une constante). Ces fonctions construisent l'objet correspondant au type dans l'AST. L'utilisation du design pattern Visiteur permet ainsi de spécifier les appels de fonction en fonction du type du noeud visité.

La visualisation de l'arbre syntaxique et de l'AST n'étant pas assez complète, nous avons amélioré l'interface graphique afin de pouvoir sélectionner un noeud directement dans la zone arborescente afin de régénérer l'arbre avec uniquement les noeuds fils.

La TDS est definie dans le package trad.semant.tds. La structure des classes est définie dans l'UML ci-dessous.



La classe TDSBlock représente un bloc de la table des symboles. Chaque TDSBlock possède un parent (sauf la TDS initiale dont le parent est null) et ses enfants sont stockés dans une ArrayList de TDSBlock. Pour la génération de code, des fonctions permettent de calculer le déplacement dans un bloc. Une TDSBlock est constituée de TDSEntry, qui sont définies soient comme étant des fonctions, variables, arguments ou références. Cela précise le type de passage de l'Identifier dans la TDS. On y enregistre aussi l'offset qui sera plus tard utilisé pour la génération de code. La classe TDSConstruct permet de construire la tables des symboles a partir d'un TDSBlock. Elle permet d'ajouter une entrée dans la TDS. On utilise les indices currentDepth pour déterminer la profondeur actuelle de la TDS et blockCount pour déterminer le nombre de blocs. La fonction backBlock permet d'obtenir le bloc parent du bloc courant, qui est utilisé lorsqu'on veut remonter les chaînages statiques pour la génération du code.

Erreurs traitées

Erreurs lexicales / syntaxiques

Nous avons laissé à ANTLR4 la responsabilité de gérer les erreurs lexicales et sémantiques. Il est à noter que celle-ci est bien meilleure que dans ANTLR3.

Erreurs sémantiques

Déclarations

- Identificateur utilisé sans avoir été préalablement déclaré
- Plusieurs identificateurs de même nom déclarés au sein du même bloc
- Bornes non valides dans la déclaration d'un tableau

Cohérence des types

- Type de la valeur retournée ne correspond pas à la définition de la fonction
- Opérandes de type incompatible avec l'opérateur (additionner un entier et un booléen, comparer (>, <, <=, >=) des booléens, faire la négation logique d'un entier, etc.)
- Opérande de gauche et de droite de type incompatible pour les tests d'égalité/de non égalité
- Type différent à gauche et à droite d'une affectation
- Condition de test (if, while) de type non booléen
- Argument du mauvais type
- Nombre d'arguments passés ne correspondant pas au nombre de paramètres de la fonction.
- Indice non entier lors de l'accès à un tableau.

Opérations impossibles

- Indexation d'une variable de type autre que tableau
- Passage par référence ou assignation d'une valeur non modifiable (constante, expression)

Schémas de traduction les plus pertinents

La génération de code est assurée par des méthodes *generateCode(gen)* sur chaque nœud de l'AST. Le paramètre *gen* représente un contexte de génération et permet :

- d'ajouter du code assembleur en sortie
- de réserver, libérer les registres et de savoir lesquels sont utilisés à un instant donné de la génération
- de garder une référence vers la table des symboles du bloc courant
- de réserver des noms d'étiquettes assembleur qui sont uniques en fournissant un préfixe auquel sera adjoint un numéro (if_0, if_1…)

La méthode *generateCode()* a pour responsabilité de produire un code assembleur correspondant au nœud de l'AST dans laquelle elle se trouve. Ainsi, la méthode *IntConstant.generateCode()* stocke dans le dernier registre alloué une valeur entière constante :

```
public void generateCode(CodeGen gen) {
    gen.enter("int constant");
    if (value >= -128 && value <= 127)
        gen.emit("ldq " + value + ", " + gen.getRegister(0));
    else
        gen.emit("ldw " + gen.getRegister(0) + ", #" + value);
    gen.leave("int constant");
}
```

Illustrons avec quelques autres exemples.
Pour un if, on a :

```
public void generateCode(CodeGen gen) {
    gen.enter("if");
    String reg = gen.allocateRegister();
    expr.generateCode(gen);

    String end_label = gen.getUniqueLabel("if_end");
    String else_label = gen.getUniqueLabel("if_else");

    // permet de mettre à jour SR depuis la valeur du registre
    gen.emit("tst " + reg);

    // saut à la clause 'else' (ou à la fin du if si pas de else) si condition non vérifiée
    // c'est à dire si la valeur de test dans le registre est égale à 0
    gen.emit("jeq #" + (instr2 != null ? else_label : end_label) + "-$-2");

    instr1.generateCode(gen);

    if (instr2 != null) {
        gen.emit("jea @" + end_label);
        gen.setLabel(else_label);
        instr2.generateCode(gen);
    }
    gen.setLabel(end_label);
    gen.leave("if");
}
```

Dans cet exemple, **expr** désigne le nœud de l'AST correspondant à l'expression booléenne du test. En appelant sa méthode *generateCode()*, on génère le code produisant le résultat (1 ou 0)

de l'expression en le stockant dans le dernier registre alloué (c'est à dire celui stocké dans la variable **reg**).
La méthode *setLabel()* permet de définir l'étiquette qui sera attachée à la prochaine instruction générée (appel à *emit()*)

Ainsi, le code généré par ce nœud pour l'entrée if (x > 10) then { // code si vrai } else { // code si faux } sera :

```
    // Enter: if
        // Enter: GREATER_THAN
            // Enter: idf: x
                adi bp, r2, #var_0_x
                ldw r2, (r2)
            // Exit: idf: x
            // Enter: int constant
                ldq 10, r3
            // Exit: int constant
            // r2 gt r3
            ldq -1, r1
            cmp r2, r3
            bgt 2
            ldq 0, r1
        // Exit: GREATER_THAN
        tst r1
        jeq #if_else_0-$-2
        jea @if_end_0
        if_else_0  nop // Alias label
    // Exit: if
```

Pour un identificateur, on a :

```
public void generateCode(CodeGen gen, boolean wantAddress) {
    gen.enter("idf: " + this);
    String base = "bp";
    if (entry.getDepth() != gen.getCurrentBlock().getDepth()) {
        gen.emit("ldw wr, (bp)");
        base = "wr";
    }
    gen.emit("adi " + base + ", " + gen.getRegister(0) + ", #var_" + entry.getBlockNumber() + "_" +
entry.getIdentifier());
    if (entry.isRef()) {
        gen.emit("ldw " + gen.getRegister(0) + ", " + "(" + gen.getRegister(0) + ")");
    }
    if (!wantAddress) {
        gen.emit("ldw " + gen.getRegister(0) + ", " + "(" + gen.getRegister(0) + ")");
    }
    gen.leave("idf: " + this);
}
```

Pour un déclaration de fonction :

```
@Override
public void generateCode(CodeGen gen) {
    TDSBlock previousBlock = gen.getCurrentBlock();
    gen.setCurrentBlock(tds);

    gen.beginScope("Fonction : " + id);

    instructionList.generateCode(gen);

    gen.setLabel("fun_" + id);
    gen.saveRegisters();
    gen.genProlog();
    gen.genVarList();

    gen.setLabel("fun_end_" + id);
    gen.genEpilog();
    gen.restoreRegisters();
    gen.emitFooter("rts");

    gen.endScope("Fonction : " + id);
    gen.setCurrentBlock(previousBlock);
}
```

On introduit ici de nouvelles méthodes :

- beginScope() : démarre un nouveau bloc (fonction) associé à une liste de registres utilisés (vide au début), une liste d'instructions assembleurs (subdivisée en début, corps, et fin)
- genHeader() : produit du code assembleur et l'ajoute dans l'entête du bloc.
- genFooter() : idem mais à la fin du bloc
- saveRegisters() : produit le code assembleur permettant de sauvegarder les registres utilisés. Elle est donc appelée après avoir généré le code des instructions mais ajoute le code de sauvegarde avant le code de l'instruction (avec emitHeader())
- restoreRegisters() : idem mais pour la restauration des registres (avec emitFooter())
- genEpilog() : génère l'épilogue d'une fonction (abandon variables locales et restauration du registre de base)
- genProlog() : génère le prologue d'une fonction (réservation variables locales, sauvegarde de la base de l'appelant)
- genVarList() : génère les instructions réservant la place nécessaire sur la pile ainsi que des directives "equ" associant à un nom de variable son déplacement
- endScope() : termine le dernier bloc ouvert et rassemble les instructions assembleur du bloc.

Programmes d’essais et limites rencontrées

Programme incorrect

```
1.  /* Ce programme montre toutes les erreurs sémantiques générées par notre compilateur */
2.
3.  program erreurs
4.  var myvoid : void;                      /* variable de type void */
5.  var dup, dup : bool;                    /* double déclaration */
6.  var dup : int;
7.  var array1 : array[1..0] of int;        /* bornes invalides */
8.  var array2 : array[-1..2, 0..3] of int; /* ok */
9.  var int1, int2, int3 : int;
10. var bool1, bool2, bool3 : int;
11.
12. function A() : void { return 10 }       /* Types de retour */
13. function B() : int { write 42 }
14. function C() : int { return false }
15.
16. function C() : void { }                 /* double déclaration */
17. function dup() : void { }               /* conflit avec la variable */
18.
19. function D(a : int) : int
20. var a : int;                            /* double déclaration */
21. var a : bool; { return a * a }
22.
23. function F(ref b : int) : void { b = 42; int1 = 2 }
24.
25. {
26.     /* Incohérences de types */
27.     int1 = true;
28.     bool1 = 10;
29.
30.     int1 = 1 + true;
31.     bool1 = false or 1;
32.     bool1 = -true;
33.
34.     int1 = bool1;
35.
36.     /* Fonctions */
37.     int1 = A();                         /* expression void */
38.     int1 = B(42);                       /* argument en trop */
39.     D();                                /* argument manquant */
40.     D(true);                            /* argument de type incorrect */
41.     F(42);                              /* référence sur constante */
```

```
42.     F(int1 + int2);
43.
44.     /* Tableaux */
45.     array1 = 42;                        /* tableau != entier */
46.     int1[0] = 42;                       /* non indexable */
47.     int1 = array1[-1, 0];               /* indice superflu */
48.     int1 = array2[0];                   /* indice manquant */
49.     int1 = array2[false, 0];            /* indice non entier */
50.
51.     if (42) then { } else { }           /* test non booléen */
52. }
```

On obtient en sortie :

```
4:1     erreur: void est invalide pour une variable
4:5     erreur: identificateur non déclaré : myvoid
5:10    erreur: variable en conflit avec un identificateur existant :  dup
6:5     erreur: variable en conflit avec un identificateur existant :  dup
7:20    erreur: la deuxième borne doit être supérieure à la première
12:23   erreur: l'expression retournée ne correspond pas au type de retour de la fonction : VOID
13:1    erreur: fonction B: la valeur de retour doit être de type INT
14:22   erreur: l'expression retournée ne correspond pas au type de retour de la fonction : INT
14:1    erreur: fonction C: la valeur de retour doit être de type INT
16:10   erreur: nom de fonction en conflit avec un identificateur existant : C
17:10   erreur: nom de fonction en conflit avec un identificateur existant : dup
20:5    erreur: variable en conflit avec un identificateur existant :  a
21:5    erreur: variable en conflit avec un identificateur existant :  a
27:5    erreur: type différent à gauche et à droite de l'affectation
30:16   erreur: opérateur "+" : l'opérande de droite n'est pas de type entier
31:13   erreur: opérateur "or" : l'opérande de droite n'est pas de type booléen
31:5    erreur: type différent à gauche et à droite de l'affectation
32:13   erreur: opérateur "-" appliqué sur un type autre qu'entier
32:5    erreur: type différent à gauche et à droite de l'affectation
37:5    erreur: type différent à gauche et à droite de l'affectation
38:14   erreur: argument 1 superflu
39:7    erreur: argument 1 manquant
40:7    erreur: l'argument 1 de la fonction n'a pas le type attendu
41:7    erreur: passage par référence impossible car la partie gauche ne peut pas être assignée
42:7    erreur: passage par référence impossible car la partie gauche ne peut pas être assignée
45:5    erreur: assignation impossible
46:5    erreur: variable non indexable (n'est pas un tableau)
47:23   erreur: indice 2 superflu
48:20   erreur: indice 2 manquant
49:19   erreur: l'indice de tableau 1 n'est pas de type INT
51:5    erreur: condition de type non booléen
```

Programmes fonctionnels

```
/* Ce programme imprime le texte de la chanson "99 bottles of beer"
 * Il est volontairement rendu complexe pour tester le compilateur
 */
program ninetyninebottles
var bottles : int;

function bottlesofbeer(n: int, title: bool, onthewall: bool) : void {
    if (n == 0) then {
        if (title) then
            write "No more";
        else
            write "no more";
    } else {
        write n;
    }

    if (n != 1) then
        write " bottles of beer";
    else
        write " bottle of beer";

    if (onthewall) then
        write " on the wall";
}

function firstsentence(n: int) : void {
    bottlesofbeer(n, true, true);
    write ", ";
    bottlesofbeer(n, false, false);
    writeln ".";
}

function secondsentence(ref n: int) : void {
    if (n == 0) then
        write "Go to the store and buy some more, ";
    else
        write "Take one down and pass it around, ";

    n = n - 1;
    if (n < 0) then n = 99;
    bottlesofbeer(n, false, true);
    writeln "."
}
{
    bottles = 99;

    while (bottles >= 0) do {
        firstsentence(bottles);
        secondsentence(bottles);
        writeln;

        if (bottles == 99) then {
            bottles = -1;
        }
    }
}
```

Répartition du travail

Il est à noter que toutes les phases de réflexion ont été réalisés avec tous les membres du groupe dans un soucis de collaboration active où chacun à pu apporter sa vision sur le projet.

	Benjamin	Thomas	David	Mickaël
Heures travaillées	35h	45h	45h	55h
Tâches effectuées	- construction de la grammaire - analyse lexicale - swing pour l'AST - génération de code	- construction de la grammaire - analyse lexicale - TDS - génération de code	- construction de la grammaire - analyse lexicale - arbre syntaxique - génération de code	- construction de la grammaire - analyse lexicale - arbre syntaxique - génération de code

Conclusion

Ce projet fût pour nous l'occasion de réaliser un véritable compilateur du langage LEAC vers du code assembleur microPIUP.
Il nous a également appris à utiliser l'outil ANTLR dans le but de générer un analyseur syntaxique.

Concernant le développement, nous avons eu quelques difficultés à appréhender l'outil ANTLR au début et à définir notre diagramme de classe.

La formation du groupe était également adaptée puisque nous avons pu développer nos compétences en programmation Java, utiliser les compétences complémentaires de David en compilation puisqu'il est en spécialité Ingénierie Logiciel et faire partager nos compétences en architecture des ordinateurs et en micro-programmation pour les autres membres du groupe qui sont en Logiciels Embarqués.
Nous avons amélioré nos compétences en gestion de projet avec des compétences propres à chacun.