

RAPPORT PROJET DE COMPILATION

SOMMAIRE

Introduction.....	3
La grammaire du langage.....	3
Aspects syntaxiques et sémantiques	3
Langage microC.....	4
L'arbre abstrait	4
La table des symboles	7
Contenu de la table des symboles	7
Les limites de notre table des symboles.....	9
Le contrôle sémantique.....	9
Erreurs traitées par le compilateur	9
Exemple de code qui génère des erreurs sémantiques	10
Résultats affichés par le compilateur	11
Les schémas de traduction	12
Programmation du compilateur	18
Conception générale.....	18
Génération de code et optimisations	18
Utilisation du compilateur	19
Les jeux d'essais	19
Le calcul du PGCD	20
Le tri selection.....	22
Organisation.....	24
Répartition des tâches	24
Estimation du temps.....	24
Conclusion	25
Annexes	26

I. Introduction

Dans le cadre du projet de compilation, nous avons du écrire un compilateur du langage microC produisant en sortie du code assembleur microPIUP/ASM.

Pour réaliser ce compilateur, nous avons du d'abord écrire une grammaire reconnaissant le langage microC, pour ensuite créer les règles de construction de l'Arbre Abstrait. Une fois l'Arbre Abstrait créé, nous nous sommes occupés de la création de la table des symboles, des contrôles sémantiques et enfin de la génération de code.

L'ensemble de ces étapes sera détaillé dans la suite de ce rapport.

II. La grammaire du langage

a. Aspects syntaxiques et sémantiques

Le langage microC créé permet de reconnaître les constructions suivantes :

Instruction	Langage microC
Les déclarations de variables, de tableaux, de pointeurs et de pointeurs de pointeurs.	<code>int var;</code> <code>int var = 0;</code> <code>int var1, var2;</code> <code>char var;</code> <code>char var = 'c';</code> <code>int tab[5];</code> <code>char tab[3] = {'a', 'b', 'c'};</code> <code>int *ptr;</code> <code>int *ptr1 = &a;</code> <code>int **ptr_ptr;</code>
Les conditions	<code>if (condition) {}</code> <code>[else {}]</code>
Les itérations	<code>while (condition) {}</code> <code>for (initialisation;condition;expression) {}</code>
Les expressions reconnues	<code>+, -, *, /, !, &&, , ==, !=, <, <=, >, >=</code>
Les retours	<code>return (expression);</code>
Les déclaration de fonctions/ procédures	<code>(int char) nom_fonction((int char) var , [...]) {}</code> <code>void nom_procedure((int char) var , [...]) {}</code>

Au début du projet nous avons essayé de rendre notre grammaire aussi proche que possible du langage C. Notre langage microC permet ainsi de reconnaître des expressions contractées (++ , -- , += , ...), des blocs sans accolades lorsqu'il n'y a qu'une instruction.

b. Langage microC

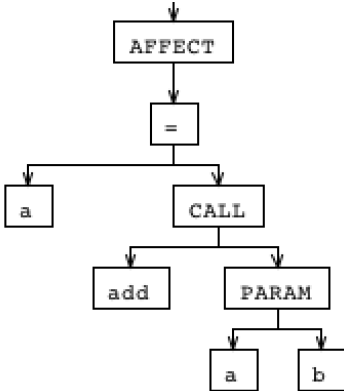
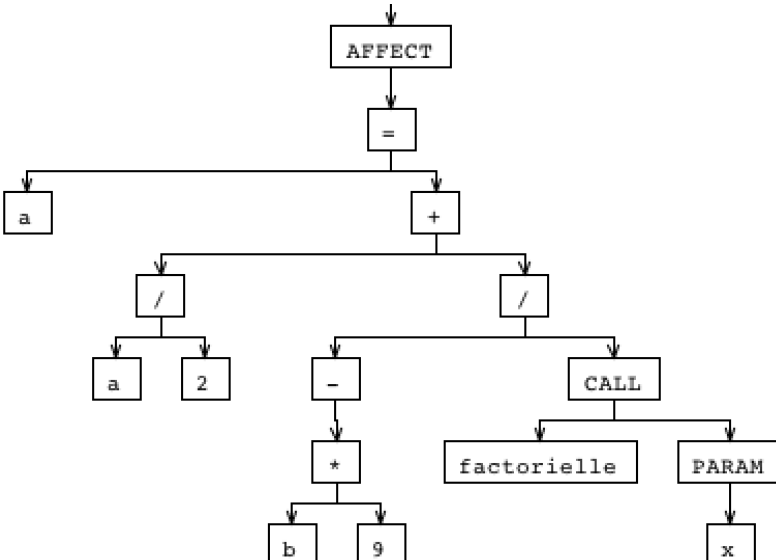
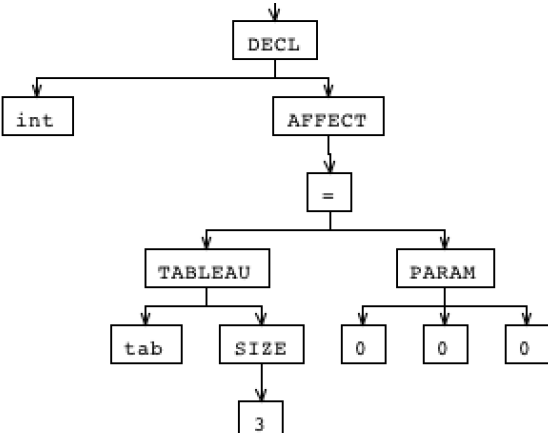
La grammaire du langage microC se trouve en annexe.

III. L'arbre abstrait

La création de l'arbre abstrait est la dernière étape avant de pouvoir générer du code java. Il est très important de simplifier au maximum l'arbre abstrait afin de pouvoir le parcourir facilement en JAVA.

● Quelques exemples :

Langage microC	AST
int main() {}	<pre> graph TD nil[nil] --> MAIN[MAIN] MAIN --> int[int] MAIN --> BLOC[BLOC] </pre>
int add(int x, int y) { return x+y; }	<pre> graph TD FONCT[FONCT] --> add[add] FONCT --> int1[int] FONCT --> PARAM[PARAM] FONCT --> BLOC[BLOC] PARAM --> DECL1[DECL] PARAM --> DECL2[DECL] DECL1 --> int2[int] DECL1 --> x1[x] DECL2 --> int3[int] DECL2 --> y[y] BLOC --> return[return] return --> plus[+] plus --> x2[x] plus --> y2[y] </pre>

Langage microC	AST
a = add(a, b);	 <pre> graph TD AFFECT --> EQ[=] EQ --> a1[a] EQ --> CALL[CALL] CALL --> add[add] CALL --> PARAM1[PARAM] PARAM1 --> a2[a] PARAM1 --> b[b] </pre>
a=(a/2)+-(b*9)/factorielle(x);	 <pre> graph TD AFFECT --> EQ[=] EQ --> a1[a] EQ --> PLUS[+] PLUS --> DIV1[/] PLUS --> MINUS[-] PLUS --> DIV2[/] DIV1 --> a2[a] DIV1 --> 2[2] MINUS --> MULT[*] MULT --> b[b] MULT --> 9[9] DIV2 --> CALL[CALL] CALL --> factorielle[factorielle] CALL --> PARAM[PARAM] PARAM --> x[x] </pre>
int tab[3]={0,0,0};	 <pre> graph TD DECL --> int[int] DECL --> AFFECT[AFFECT] AFFECT --> EQ[=] EQ --> TABLEAU[TABLEAU] EQ --> PARAM[PARAM] TABLEAU --> tab[tab] TABLEAU --> SIZE[SIZE] SIZE --> 3[3] PARAM --> 0_1[0] PARAM --> 0_2[0] PARAM --> 0_3[0] </pre>

Langage microC	AST
<pre>for(i=0;i<3;i=i+1) tab[i]=i;</pre>	<pre> graph TD FOR[FOR] --> COND[COND] FOR --> BLOC1[BLOC] COND --> AFFECT1[AFFECT] COND --> LT[<] COND --> AFFECT2[AFFECT] AFFECT1 --> EQ1[=] EQ1 --> i1[i] EQ1 --> 0[0] LT --> i2[i] LT --> 3[3] AFFECT2 --> EQ2[=] EQ2 --> i3[i] EQ2 --> PLUS[+] PLUS --> i4[i] PLUS --> 1[1] BLOC1 --> AFFECT3[AFFECT] AFFECT3 --> EQ3[=] EQ3 --> TABLEAU[TABLEAU] EQ3 --> i5[i] TABLEAU --> tab[tab] TABLEAU --> INDICE[INDICE] INDICE --> i6[i] </pre>
<pre>if(a==b) *ptr = &a else exit()</pre>	<pre> graph TD IF[IF] --> COND[COND] IF --> BLOC1[BLOC] IF --> ELSE[ELSE] IF --> BLOC2[BLOC] COND --> EQ[==] EQ --> a[a] EQ --> b[b] BLOC1 --> AFFECT[AFFECT] AFFECT --> EQ2[=] EQ2 --> PTR[PTR] PTR --> ptr[ptr] EQ2 --> ADDRESS[ADDRESS] ADDRESS --> a2[a] ELSE --> CALL1[CALL] CALL1 --> exit1[exit] BLOC2 --> CALL2[CALL] CALL2 --> exit2[exit] CALL2 --> PARAM[PARAM] </pre>

IV. La table des symboles

La création de la table des symboles est la première procédure à réaliser en JAVA après avoir récupéré l'arbre syntaxique abstrait. Cette opération permet de centraliser toutes les informations concernant les identificateurs (de variable, fonction, etc...).

a. Contenu de la table des symboles

Il y a une table des symboles pour l'espace global et pour chaque fonction/procédure déclaré dans le programme. Ces différentes tables des symboles sont stockées sous la forme d'une liste dans une structure *HashMap* (*HashMap<String, TableSymboles> liste_tds;*). La clé pour chacune des tables des symboles est le nom de la fonction/procédure associée (dans le cas de l'espace global on utilise la clé '#global#').

La classe *TableSymboles* contient aussi un *HashMap* permettant d'associer un identificateur (la clé) à une instance de la classe *Symbole* (qui rassemble toutes les informations que l'on peut avoir besoin pour le contrôle sémantique et la génération de code)

Ainsi la classe *Symbole* contient (voir diagramme de classes en annexe) :

- **Le type** : int, char ou void.
- **Le mode** : direct, indirect (pour les pointeurs) ou double_indirect (pour les pointeurs de pointeurs).
- **La nature** : variable, tableau, paramètre de fonction, fonction ou procedure.
- **L'adresse** : représentant le décalage par rapport au pointeur de la base.
- **La taille** : la taille de la variable en fonction de son type (multiplié par le nombre d'éléments s'il s'agit d'un tableau).

- Exemple de table des symboles obtenue avec le code ci dessous :

```

1  int a,b;
2
3  int add (int x,int y){ return a+b; }
4
5  void afficheAdd (int x,int y,int res) { /*affichage de l'addition*/ }
6
7  int main(){
8      int res;
9      res = add(a,b);
10 }

```

Nom table : #global#

nom	type	mode	nature	adresse	taille
b	INT	DIRECT	VARIABLE	-4	2
a	INT	DIRECT	VARIABLE	-2	2
afficheAdd	VOID	DIRECT	PROCEDURE	0	3
add	INT	DIRECT	FONCTION	0	2
main	INT	DIRECT	FONCTION	0	0

Nom table : add

nom	type	mode	nature	adresse	taille
y	INT	DIRECT	PARAM	8	2
x	INT	DIRECT	PARAM	4	2

Nom table : afficheAdd

nom	type	mode	nature	adresse	taille
res	INT	DIRECT	PARAM	12	2
y	INT	DIRECT	PARAM	8	2
x	INT	DIRECT	PARAM	4	2

Nom table : main

nom	type	mode	nature	adresse	taille
res	INT	DIRECT	VARIABLE	-2	2

b. Les limites de notre table des symboles

Comme décrit ci-dessus, l'utilisation des *HashMaps* pour la table des symboles permet d'associer une clé (nom de la variable, de la fonction ou de la procédure) à une instance de la classe *Symbole* qui permet de décrire un certain nombre d'éléments utiles pour la table des symboles. Cette méthode de stockage est très efficace et permet d'accéder rapidement aux données associées à un identifiant (variable, fonction ou procédure). Cependant, une clé doit être unique, ce qui implique que l'on ne peut pas par exemple déclarer deux fonctions ayant le même nom avec des paramètres différents (*int fonction(int a){}* et *int fonction(int a, int b){}* ne pourront pas fonctionner dans le même programme).

Les identifiants (de variable, fonction, etc ...) étant potentiellement infinis, nous avons conclu que cette contrainte n'était pas suffisamment importante pour que l'on cherche une autre solution de stockage de notre table des symboles.

Cependant une solution possible serait de hasher les noms de fonctions/procédures en fonction de leur entête pour ainsi avoir des clés différentes pour des fonctions ayant le même nom.

V. Le contrôle sémantique

Le contrôle sémantique a été particulièrement long à réaliser, cela s'explique par le fait que notre grammaire est très permissive : en effet au début du projet nous nous sommes fixés pour objectif que notre grammaire devait pouvoir reconnaître un maximum d'écriture du langage C. Nous avons atteint notre objectif mais cela a pour défaut d'ajouter de nombreux cas à tester lors du contrôle sémantique.

a. Erreurs traitées par le compilateur

- ▶ Vérifie qu'une fonction retourne bien une valeur.
- ▶ Vérifie si une variable utilisée a bien été déclaré.
- ▶ Vérifie si une fonction/procédure appelée a bien été déclaré.
- ▶ Vérifie le nombre de paramètres passés à une fonction/procédure.
- ▶ Vérifie les types des constantes/variables passées en paramètres à une fonction/procédure.
- ▶ Vérifie si une variable est en 'overflow'.
- ▶ Vérifie les bornes lors de l'accès à un tableau.

b. Exemple de code qui génère des erreurs sémantiques

```
1 void f1(int a){
2     int x,y;
3 }
4
5 int f2(char a){
6 }
7
8 void x1(char a,int b,char c){
9     return 0;
10 }
11
12 int x2(char* a,char* b,char *c,char d){
13     return 0;
14 }
15
16 int main(){
17     int x;
18     int y;
19     char c;
20     char *p;
21     f0(a,b);
22     f1(x);
23     f1(x,y);
24     x1('a','i',1);
25     printi();
26     printi(-1);
27     printi('a');
28     printc(1);
29     printc('a');
30     exit(0);
31     x2(p,*c,&p,p);
32     x=38697;
33     y=-354454;
34     y=-4;
35     char t[3]={'a','b','c'};
36     char t[3]={'a','b','c','d'};
37     t[3]='a';
38     t[-1]='a';
39 }
```

c. Résultats affichés par le compilateur

```
Erreur: ligne 5: pas de 'return' dans la fonction 'f2'.  
Erreur: ligne 21: fonction 'f0' appelée mais non déclarée.  
Erreur: ligne 23: La fonction 'f1' utilise 1 parametre (2 parametres lors de l'appel).  
Warning: ligne 24: Le parametre 2 de la fonction 'x1' est de type 'INT' (type CHAR utilisé).  
Warning: ligne 24: Le parametre 3 de la fonction 'x1' est de type 'CHAR' (type INT utilisé).  
Erreur: ligne 25: La fonction 'printi' utilise 1 parametre (0 parametre lors de l'appel).  
Warning: ligne 27: Le parametre 1 de la fonction 'printi' est de type 'INT' (type CHAR utilisé).  
Warning: ligne 28: Le parametre 1 de la fonction 'printc' est de type 'CHAR' (type INT utilisé).  
Erreur: ligne 30: La fonction 'exit' n'utilise pas de parametre (1 parametre lors de l'appel).  
Erreur: ligne 31: Utilisation interdite de la variable 'c' (parametre 2) dans l'appel de la  
fonction 'x2'  
Warning: ligne 31: Le parametre 3 de la fonction 'x2' est de type 'CHAR *' (type CHAR ** utilisé).  
Warning: ligne 31: Le parametre 4 de la fonction 'x2' est de type 'CHAR' (type CHAR * utilisé).  
Warning: ligne 32: overflow (32767 max).  
Warning: ligne 33: overflow (32767 max).  
Erreur: ligne 36: Le tableau 't' est de taille 3 (4 valeurs pour l'initialisation).  
Erreur: ligne 37: depassement des bornes du tableau 't' [0..2].  
Erreur: ligne 38: depassement des bornes du tableau 't' [0..2].
```

Le code ne peut pas être généré! (8 errors, 9 warnings)

VI. Les schémas de traduction

Code microC	Code assembleur produit
<pre>// FONCTION MAIN() // ET DECLARATION int main() { int a; }</pre>	<pre>main_ // Début de la fonction main(). LDW SP, #STACK_ADRS // charge SP avec STACK_ADRS. LDQ NIL, BP // Initialisation de BP:BP = NIL LDQ 2, R1 // R1 = taille données locales ADQ -2, SP // décrémente le pointeur de pile SP. STW BP, (SP) // sauvegarde le contenu du registre BP // sur la pile. LDW BP, SP // charge contenu SP ds BP qui pointe // sur sa sauvegarde. SUB SP, R1, SP // réserve R1 octets sur la pile LDW SP, BP // charge SP avec contenu de BP: // abandon infos locales. LDW BP, (SP) // charge BP avec ancien BP. ADQ 2, SP // ancien BP supprimé de la pile LDW WR, #EXIT_EXC // WR = EXIT_EXC // (n° exception de EXIT) TRP WR // exécute la trappe EXIT LDW WR, #main_ // WR = main_ // (adresse du début de main) JEA (WR) // saute à l'instruction dont l'adresse // est dans WR</pre>
<pre>// AFFECTATION a=1;</pre>	<pre>LDW R1, #1 // R1 = 1 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a STW R1, (WR) // sauvegarde la valeur dans a.</pre>
<pre>// EXPRESSION a=2*(2+1);</pre>	<pre>LDW R1, #2 // R1 = 2 LDW R2, #2 // R2 = 2 LDW R3, #1 // R3 = 1 ADD R2, R3, R2 MUL R1, R2, R1 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a STW R1, (WR)</pre>

Code microC	Code assembleur produit
// <i>CONDITION</i> if(a>b) { a=5; } else { a=10; }	LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a LDW R1, (WR) // R1 = a LDW WR, BP // WR = BP ADQ -4, WR // WR pointe sur b LDW R2, (WR) // R2 = b CMP R1, R2 // Test différence JLE #Etiquette2_-\$-2 // Si différence saut à Etiquette2 // sinon exécution de la suite LDW R1, #1 // R1 = 1 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a STW R1, (WR) JMP #Etiquette1_-\$-2 // Saut à la fin du if // pour pas aller dans le else Etiquette2_ // Etiquette pour le else LDW R1, #0 // R1 = 0 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a STW R1, (WR) Etiquette1 // Etiquette pour la fin du if

Code microC	Code assembleur produit
<pre> /* AFFICHAGE D ' U N CHARACTERE */ int main() { char c; c='x'; printf(x); } </pre>	<pre> main_ [...]</pre> <p>LDW R1, #376 // R1 = 'x' (code ascii de 'x' + 256)</p> <p>LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur c STW R1, (WR)</p> <p>ADQ -2, SP // décrémente le pointeur de pile SP STW R1, (SP) // sauvegarde le contenu du registre R1 // sur la pile</p> <p>LDW R1, #printf_</p> <p>MPC WR // charge le contenu du PC dans WR ADQ 8, WR // ajoute 8 à WR: WR contient l'adresse // de retour</p> <p>ADQ -2, SP // décrémente le pointeur de pile SP STW WR, (SP) // sauvegarde l'adresse de retour sur le // sommet de pile</p> <p>JEA (R1) // saute à l'instruction d'adresse // absolue dans R1</p> <p>ADQ 1*2, SP // SP + 1 * 2 -> SP</p> <p>[...]</p> <p>printf_ // PROCEDURE afficher un caractere</p> <p>LDQ 0, R1 // R1 = taille données locales (ici 0) de // fonction appelée</p> <p>STW BP, -(SP) // empile le contenu du registre BP LDW BP, SP // charge contenu SP ds BP qui pointe // sur sa sauvegarde</p> <p>SUB SP, R1, SP // réserve R1 octets sur la pile pour la // variable locale</p> <p>LDW R0, #NIL // On met a NIL la fin de la chaine pour // arreter l'affichage</p> <p>STW R0, -(SP) LDW R0, BP // On recopie le caractere a afficher au // sommet de la pile</p> <p>ADQ 4, R0 LDW R0, (R0) STW R0, -(SP) LDW R0, SP</p> <p>TRP #WRITE_EXC // Lancement de la TRP</p> <p>STW R0, (SP)+ // On dépile les caracteres empilés STW R0, (SP)+</p> <p>LDW SP, BP // charge SP avec contenu de BP: // abandon infos locales</p> <p>LDW BP, (SP) // charge BP avec ancien BP ADQ 2, SP // ancien BP supprimé de la pile</p> <p>RTS</p>

Code microC	Code assembleur produit
<pre>// BOUCLE i=0; while(i<10) { i=i+1; } //equivalent à: for(i=0;i<10;i=i+1) { }</pre>	<pre>LDW R1, #0 // R1 = 0 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur i STW R1, (WR) Etiquette1_ // Etiquette pour le debut de la boucle LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur i LDW R1, (WR) // R1 = i LDW R2, #10 // R2 = 10 CMP R1, R2 // Test supérieur ou égal JGE #Etiquette2_-\$-2 // Si supérieur ou égal saut à Etiquette2 // sinon exécution de la suite LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur i LDW R1, (WR) // R1 = i LDW R2, #1 // R2 = 1 ADD R1, R2, R1 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur i STW R1, (WR) JMP #Etiquette1_-\$-2 // saut au début de la boucle Etiquette2 // Etiquette pour la fin de la boucle</pre>
<pre>// POINTEUR int a = 10; int *p; int p = &a;</pre>	<pre>LDW R1, #10 // R1 = 10 LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a STW R1, (WR) LDW WR, BP // WR = BP ADQ -2, WR // WR pointe sur a LDW R1, WR // R1 = &a LDW WR, BP // WR = BP ADQ -6, WR // WR pointe sur p STW R1, (WR)</pre>

Code microC	Code assembleur produit
<i>/* PAS ASSEZ DE REGISTRES DISPONIBLES */</i>	LDW R1, #1 // R1 = 1
	LDW R2, #2 // R2 = 2
	LDW R3, #3 // R3 = 3
int a =	LDW R4, #4 // R4 = 4
(1+(2+(3+(4+(5+(6+(7+(8+(9+(LDW R5, #5 // R5 = 5
(10+(11+12))))))))))));	LDW R6, #6 // R6 = 6
	LDW R7, #7 // R7 = 7
	LDW R8, #8 // R8 = 8
	LDW R9, #9 // R9 = 9
	LDW R10, #10 // R10 = 10
	ADQ -2, SP // décrémente le pointeur de pile SP
	STW R1, (SP) // sauvegarde du registre R1 car plus assez de registres dispos
	LDW R1, #11 // R1 = 11
	ADQ -2, SP // décrémente le pointeur de pile SP
	STW R2, (SP) // sauvegarde du registre R2 car plus assez de registres dispos
	LDW R2, #12 // R2 = 12
	ADD R1, R2, R1
	LDW R2, (SP) // on restaure registre R2 précédemment sauvegardé
	ADQ 2, SP // incrémente le pointeur de pile SP
	ADD R10, R1, R10
	LDW R1, (SP) // on restaure registre R1 précédemment sauvegardé
	ADQ 2, SP // incrémente le pointeur de pile SP
	ADD R9, R10, R9
	ADD R8, R9, R8
	ADD R7, R8, R7
	ADD R6, R7, R6
	ADD R5, R6, R5
	ADD R4, R5, R4
	ADD R3, R4, R3
	ADD R2, R3, R2
	ADD R1, R2, R1

Code microC	Code assembleur produit
//TABLEAU	LDQ 6, R1 // R1 = taille données locales
int tab[2]={1,2};	[...]
tab[1]=5;	
int a=tab[0];	LDW R1, #1 // R1 = 1
	LDW WR, BP // WR = BP
	ADQ -2, WR // WR pointe sur tab[0]
	STW R1, (WR) // tab[0] = R1
	LDW R1, #2 // R1 = 2
	LDW WR, BP // WR = BP
	ADQ -4, WR // WR pointe sur tab[1]
	STW R1, (WR) // tab[1] = R1
	LDW R1, #5 // R1 = 5
	LDW WR, BP // WR = BP
	ADQ -4, WR // WR pointe sur tab[1]
	STW R1, (WR) // tab[1] = R1
	LDW WR, BP // WR = BP
	ADQ -2, WR // WR pointe sur tab[0]
	LDW R1, (WR) // R1 = tab[0]
	LDW WR, BP // WR = BP
	ADQ -6, WR // WR pointe sur a
	STW R1, (WR)

VII. Programmation du compilateur

Les sources du projet sont disponibles sur le dépôt subversion Redmine :

<http://redmine.esial.uhp-nancy.fr/svn/frantz4u-guerci2u>

a. Conception générale

Le compilateur que nous proposons comporte plusieurs modules séparés dans des classes distinctes (voir le diagramme de classe en **annexe**) :

► **Le module principal (Compilateur.java) :**

C'est dans ce module que l'arbre abstrait est construit, ensuite l'analyse est lancée (tables des symboles, contrôles sémantiques), pour enfin exécuter la génération de code.

► **Le module de construction des tables de symboles (TableSymboles.java) :**

C'est dans ce module que toutes les tables de symboles (pour les variables globales, le main, etc...) sont construites et rendues accessibles dans tous les autres modules.

► **Le module de contrôles sémantiques (ControleSemantique.java) :**

C'est dans ce module que sont effectués tous les contrôles sémantiques (décrits dans la section « Contrôle sémantique »).

► **Le module génération de code (GenerateurDeCode.java) :**

C'est dans ce module que toute la génération du code Assembleur est effectuée.

b. Génération de code et optimisations

- **Génération par noeuds** : Lors du parcours de l'AST, À chaque noeud est associé une fonction spécifique pour la génération de code (Affectation, Appel de fonction, etc...).
- **Appel "automatique" de fonctions** : Toujours lors du parcours de l'AST, l'appel de la fonction correspondante au noeud parcouru se fait tout seul grâce à l'implémentation d'une *HashMap* de fonctions qui pour une clé donnée (par exemple IF ou FOR) associe une fonction spécifique de génération de code (par exemple 'genererIf' ou 'genererFor').

- **Utilisation des registres optimisée** : Lorsqu'il faut évaluer des expressions (affectation d'une expression arithmétique ou évaluation d'une condition), nous utilisons le maximum de registres disponibles et nous utilisons la pile uniquement quand il n'y a plus de registres disponibles.
- **Évaluation paresseuse des expressions (optimisation)** : Lorsque que le compilateur doit évaluer une condition comportant des "&&" et/ou des "||", il s'arrête automatiquement lorsqu'il devient logique que la condition ne sera pas vérifiée.

Par exemple : `if(I!=1 && I==1)`

Quand le compilateur va évaluer "`I!=1`", ça sera faux, or on sait que faux ET (vrai OU faux) sera toujours faux, donc le compilateur ne cherchera pas à évaluer "`I==I`", il évaluera donc seulement le nécessaire.

VIII.Utilisation du compilateur

```
java -jar compilateur.jar chemin/source.c [-i]
```

L'option -i permet d'afficher des informations sur les différents processus de la compilation (l'arbre syntaxique, la table des symboles, les erreurs sémantiques, la génération de code, etc...)

Si la compilation se termine avec succès un fichier exécutable microPIUP est créé à coté du fichier source passé en paramètre : **chemin/source.iup**.

Pour exécuter le fichier microPIUP créé il faut lancer microPIUP grâce à la commande **java -jar microPIUP4.jar -sim&** dans le terminal. Une fois lancé, ouvrir l'exécutable en cliquant sur le bouton « *Charger un fichier* » et cliquer sur « *Exécuter le programme* » (l'affichage produit par le programme est visible dans l'onglet « *Périphériques* », il faut s'assurer que la case « *Avec primitives système* » soit bien cochée).

IX. Les jeux d'essais

Pour montrer le bon fonctionnement de notre compilateur, nous avons décidé d'implémenter deux programmes :

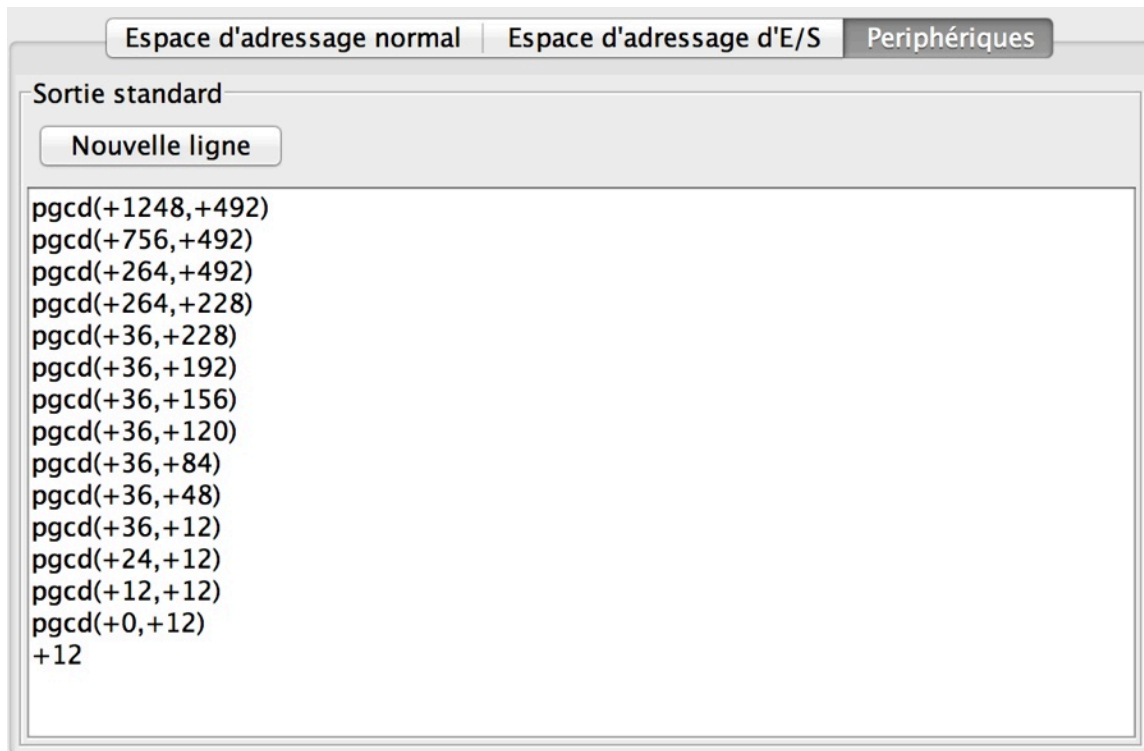
a. Le calcul du PGCD

Cet exemple permet de mettre en évidence le bon fonctionnement des déclarations/initialisations de variables, des appels de fonctions, des appels récursifs, des affichages sur la sortie standard, des conditions et des calculs simples (soustractions).

● Code :

```
1  int pgcd(int p, int q){
2      int r;
3      affiche(p,q);
4      if(p==0)
5          r=q;
6      else
7          if(q==0)
8              r=p;
9          else
10             if(q<=p){
11                 p=p-q;
12                 r=pgcd(p,q);
13             }
14             else{
15                 q=q-p;
16                 r=pgcd(p,q);
17             }
18         return r;
19     }
20
21 void affiche(int x, int y){
22     printf('p');
23     printf('g');
24     printf('c');
25     printf('d');
26     printf('(');
27     printf(x);
28     printf(',');
29     printf(y);
30     printf(')');
31     printf('\n');
32 }
33
34 int main(){
35     int a;
36     int b;
37     int res;
38     a=1248;
39     b=492;
40     res=pgcd(a,b);
41     printf(res);
42     printf('\n');
43 }
```

● **Affichage :**



b. Le tri sélection

Cet exemple permet de mettre en évidence le bon fonctionnement des déclarations/initialisations des variables et des tableaux, des appels de fonctions/procédures, des affichages sur la sortie standard, des conditions et des boucles.

● Code :

```
1  int tab[15] = {54,34,0,12,49,4,2,5,20,67,59,90,101,23,99};
2
3  void triSelection(int n){
4      int i = 0;
5      int j = 0;
6      int min,tmp;
7      for(i=0; i<n-1; i=i+1){
8          min = i;
9          for(j=i+1; j<n; j=j+1)
10             if(tab[j] < tab[min])
11                 min = j;
12             if(min != i) {
13                 tmp = tab[i];
14                 tab[i] = tab[min];
15                 tab[min] = tmp;
16             }
17         }
18     }
19
20 void affiche(int esttrie){
21     int i;
22     if(esttrie==1){
23         printf('n');
24         printf('o');
25         printf('n');
26         printf(' ');
27     }
28     printf('t');
29     printf('r');
30     printf('i');
31     printf('e');
32     printf(':');
33     printf('\n');
34     for(i=0; i<15; i=i+1){
35         printi(tab[i]);
36     }
37     printf('\n');
38     printf('\n');
39 }
40
41 void main(){
42     affiche(1);
43     triSelection(15);
44     affiche(0);
45 }
```

● **Affichage :**

The screenshot shows a software window with three tabs at the top: 'Espace d'adressage normal', 'Espace d'adressage d'E/S', and 'Périphériques'. The 'Périphériques' tab is selected. Below the tabs, the text 'Sortie standard' is displayed. Underneath, there is a button labeled 'Nouvelle ligne'. The main content area contains two lines of text: 'non trie:' followed by the sequence '+54+34+0+12+49+4+2+5+20+67+59+90+101+23+99', and 'trie:' followed by the sequence '+0+2+4+5+12+20+23+34+49+54+59+67+90+99+101'.

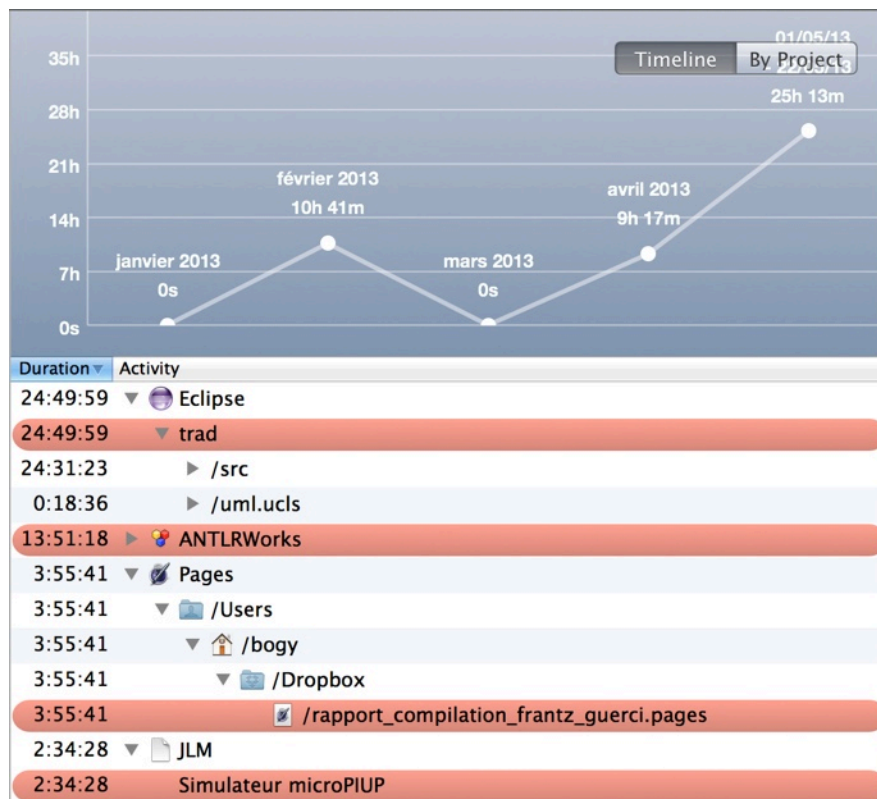
X. Organisation

a. Répartition des tâches

L'ensemble du projet a été équitablement réparti, nous avons tous les deux travaillé sur chacune des étapes du projet.

b. Estimation du temps

Nous avons consacré 15 heures à l'utilisation du logiciel ANTLRWorks (pour réaliser la grammaire), 50 heures pour la programmation du compilateur sur Eclipse, au moins 5 heures de test sur microPIUP et enfin 8 heures pour la rédaction du rapport. Soit un total d'environ 80 heures de travail.



*Aperçu du logiciel Mac «Timing» permettant de suivre le nombre d'heures passées sur un projet.
Les durées présentées ci-dessus sont uniquement pour l'ordinateur de Richard GUERCI.
(On peut estimer le temps total du projet en multipliant ces durées par deux.)*

XI. Conclusion

Ce projet nous a permis de mettre en pratique les connaissances que nous avons apprises dans le module de Traduction 1 et 2. Nous avons trouvé intéressant de découvrir et de connaître maintenant les étapes permettant de concevoir un compilateur de A à Z.

La réalisation de ce compilateur nous a également permis de comprendre l'importance de construire un arbre abstrait que l'on peut parcourir facilement pour effectuer des contrôles sémantiques et pour générer plus aisément le code assembleur en sortie.

Bien que les difficultés rencontrées et les bogues aient été nombreux, notre compilateur obtenu est fonctionnel et permet de réaliser toutes les instructions demandées dans le sujet.

Pour conclure, le projet Compilation a été une bonne expérience, vu les nombreuses compétences acquises par chacun de nous.

sources :

<http://www.antlr.org.cn/07164437662.pdf> «The Definitive ANTLR Reference»

http://www-verimag.imag.fr/~jhenry/teaching/c7_6pages.pdf (page 45)

Annexes

● Grammaire microC

```
grammar microC_AST;
options {
    backtrack=false;
    output=AST;
    ASTLabelType=CommonTree;
}
tokens {
    MAIN;
    BLOC;
    COND;
    FONCT;
    PROC;
    PTR;
    ADDRESS;
    WHILE;
    FOR;
    IF;
    ELSE;
    SIZE;
    CALL;
    PARAM;
    DECL;
    AFFECT;
    TABLEAU;
    INDICE;
    CARACTERE;
} // token imaginaire

//debut
programme      :
    (declaration FIN_INSTRUCTION!|affectation FIN_INSTRUCTION!|fonction|procedure)* main
    ;

main           :
    type_all 'main' '(' ')' bloc_complexe -> ^(MAIN type_all bloc_complexe)
    ;

//les blocs
bloc           :
    bloc_simple
    |
    bloc_complexe
    ;

bloc_simple    :
    instruction -> ^(BLOC instruction)
    ;

bloc_complexe  :
    '{' instruction* '}' -> ^(BLOC instruction*)
    ;
```

```

//les instructions
instruction
:      condition
|      boucle
|      affectation FIN_INSTRUCTION!
|      declaration FIN_INSTRUCTION!
|      appel_fonction_procedure FIN_INSTRUCTION!
|      retour FIN_INSTRUCTION!
;

//declaration de variable
declaration
:      type_affect var ( ',' var)* -> ^(DECL type_affect var)+
|      type_affect affectation_bis -> ^(DECL type_affect affectation_bis)
;

var      :      IDENTIFIANT tableau -> ^(TABLEAU IDENTIFIANT tableau)
|      IDENTIFIANT^
;

//affectation
affectation
:      IDENTIFIANT operateur_affectation expression -> ^(AFFECT ^(operateur_affectation
IDENTIFIANT expression))
|      pointeur operateur_affectation expression -> ^(AFFECT ^(operateur_affectation ^(PTR
pointeur) expression))
|      acces_tableau operateur_affectation expression -> ^(AFFECT ^(operateur_affectation
acces_tableau expression))
;

affectation_bis
:      IDENTIFIANT '=' expression -> ^(AFFECT ^('=' IDENTIFIANT expression))
|      IDENTIFIANT tableau '=' '{parametres_appel}' -> ^(AFFECT ^('=' ^(TABLEAU
IDENTIFIANT tableau) parametres_appel))
;

tableau  :      '[' CHIFFRES ']' -> ^(SIZE CHIFFRES)
;

operateur_affectation
:      '='
|      '*='
|      '/='
|      '+='
|      '-='
;

//Appel de fonction et procedure
appel_fonction_procedure
:      IDENTIFIANT '(' parametres_appel ')' -> ^(CALL IDENTIFIANT parametres_appel)
;

```

```

//Liste de paramètres d'appel
parametres_appel
:      (expression)? (' (expression))* -> ^(PARAM expression*)
;

//les conditionnelles : instruction if ... else ...
condition
:      'if' '(' expression ')' bloc (options {greedy=true;} :else' bloc)? ->^(IF ^(COND expression)
bloc (ELSE bloc)?)
;

//Les iteration → à l'IDENTIFIANT de for et while
boucle
:      'while' '(' expression ')' bloc -> ^(WHILE ^(COND expression) bloc)
|      'for' '(' affectation ';' expression ';' affectation ')' bloc -> ^(FOR ^(COND affectation
expression affectation) bloc)
;

//Fonction et procedure
fonction :      type_affect IDENTIFIANT '(' parametres_effectifs ')' bloc_complexe -> ^(FONCT
IDENTIFIANT type_affect parametres_effectifs bloc_complexe)
;

procedure :      'void' IDENTIFIANT '(' parametres_effectifs ')' bloc_complexe -> ^(PROC
IDENTIFIANT parametres_effectifs bloc_complexe)
;

parametres_effectifs
:      parametre? (' parametre)* -> ^(PARAM (parametre)* )
;

parametre :      type_affect IDENTIFIANT -> ^(DECL type_affect IDENTIFIANT)
;

//retour de fonction
retour :      'return'^
|      'return'^ expression
;

//Types
type_int :      'int'
|      'char'
;

type_ptr :      'int' '*' -> ^(PTR 'int')
|      'char' '*' -> ^(PTR 'char')
|      'int' '*' '*' -> ^(PTR ^(PTR 'int'))
|      'char' '*' '*' -> ^(PTR ^(PTR 'char'))
;

type_affect :      type_int|type_ptr
;

type_all :      type_int|'void'
;

```

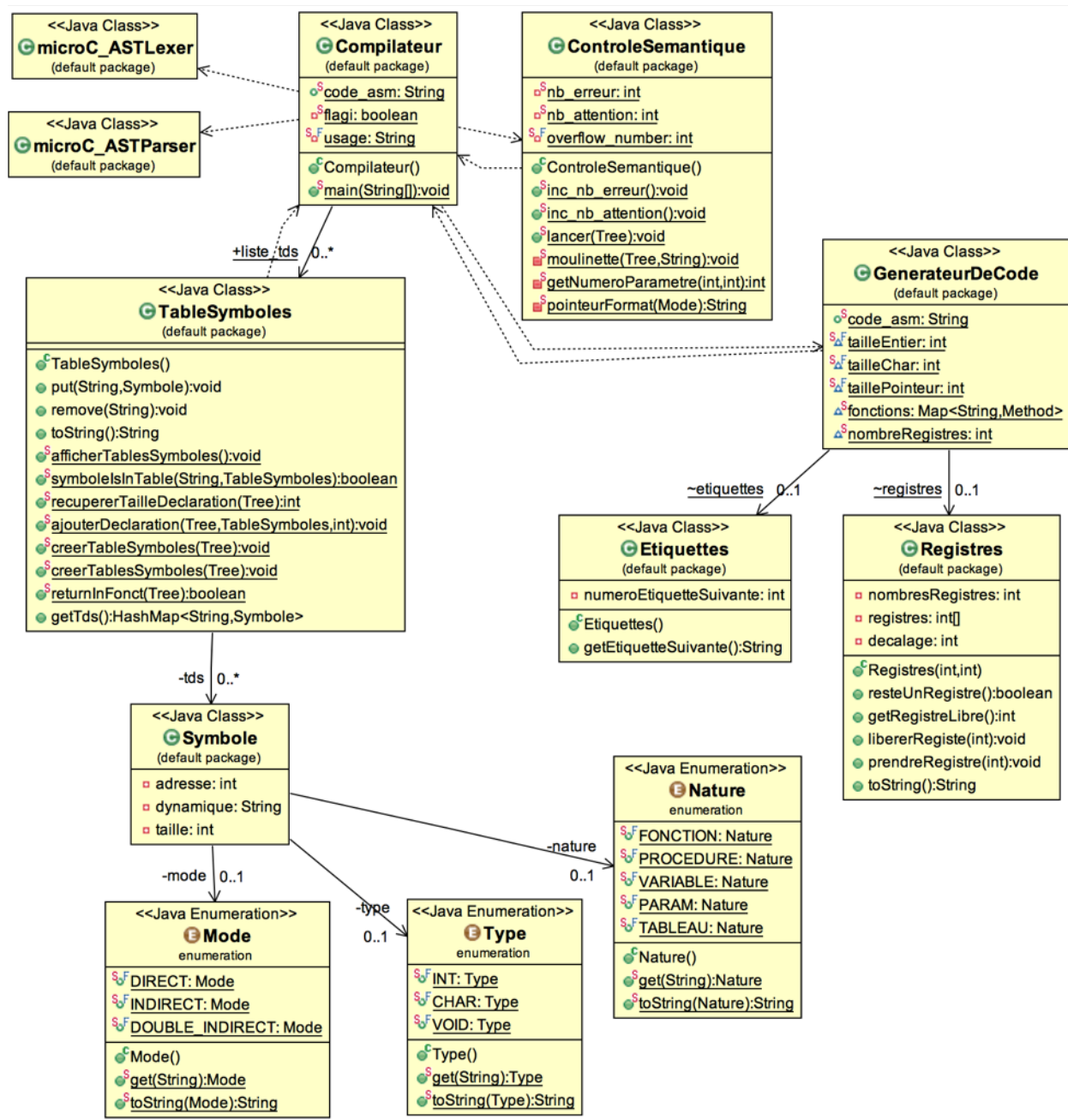
```

adresse :      '&'! ((IDENTIFIANT|pointeur)|'!(IDENTIFIANT|pointeur))'!)
;

IDENTIFIANT   :      LETTRE (LETTRE|CHIFFRE)*;
fragment
LETTRE        :      ('a'..'z'|'A'..'Z');
LETTRE_QUOTE  :      "\" (LETTRE|'|#'|@'|,|'|/|'|?|'|!|'|'(|'|)'|'|_|'|+|'|%|'|:|'|<|'|>|'|=|'|\"'|*|'|$|
CHIFFRE       :      \"\\n|\"\\t|\"\\0') \" {setText(getText().substring(1, getText().length()-1));} ;
fragment
CHIFFRE       :      '0'..'9';
CHIFFRES      :      CHIFFRE+;
COMMENTAIRE   :      '//' ( options {greedy=false;} : . )* \"n\" {$channel=HIDDEN;};
COMMENTAIRES  :      '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;};
FIN_INSTRUCTION :      '!' ;
WS            :      (' '|\"r\"|\"t\"|\"u000C\"|\"n\") {$channel=HIDDEN;};

```

● Diagramme de classe du projet



Note : Certains attributs, méthodes et relations ne sont pas affichés pour une meilleure lecture du diagramme.