Rapport

De

Projet

Preparé Par :

Najib Aghenda
Laetitia Huret
Damien Vantourout
Maylis Willaime–Angonin

Projet :
Compilation

Instructeur :
Suzanne Collin

Date :
18 Mai 2018

Table des matières

Introduction

Pour rappel, l'objectif de ce projet était d'écrire un compilateur pour le langage de programmation Mini-Rust, version simplifiée du langage Rust développé par Mozilla et fonctionnant sur le simulateur CISC créé par Alexandre Parodi, MicroPIUP. Pour ce faire, nous avons eu recours à ANTLR pour la spécification de la grammaire et la génération de l'arbre syntaxique, au langage Java pour réaliser l'analyse sémantique, ainsi que pour générer le code assembleur au format MicroPIUP / ASM. Pour la gestion des versions et le travail en équipe, nous avons utilisé git et déposé tous nos fichiers sur le dépôt gitlab de TELECOM Nancy.

Ce document présente la spécification de la grammaire, la mise en place de l'analyse lexicale et syntaxique puis décrit la création et l'implémentation de la table des symboles et des contrôles sémantiques. Il présente ensuite la génération du code assembleur. Les annexes présentent par ailleurs la grammaire du langage, des schémas de traduction, des fichiers permettant de tester le compilateur ainsi que la répartition des tâches entre les membres du groupe.

Ce projet a été réalisé par Laetitia Huret, Maylis Willaime–Angonin, Najib Aghenda et Damien Vantourout et a été encadré par Mme. Collin Suzanne.

1 Spécification de grammaire du langage Mini-Rust

1.1 La grammaire sous ANTLR

Afin de spécifier la grammaire du langage sous ANTLR, nous nous sommes référés à la syntaxe du langage qui nous était proposée et que l'on peut voir en figure 1.

```
FICHIER      ⟶  DECL*
DECL         ⟶  DECL_FUNC | DECL_STRUCT
DECL_STRUCT  ⟶  struct idf { [ idf : TYPE (, idf : TYPE)* ] }
DECL_FUNC    ⟶  fn idf ( [ ARGUMENT (, ARGUMENT)* ] ) [ -> TYPE ] BLOC
TYPE         ⟶  idf | Vec < TYPE > | & TYPE | i32 | bool
ARGUMENT     ⟶  idf : TYPE
BLOC         ⟶  { INSTRUCTION* [EXPR] }
INSTRUCTION  ⟶  ;
                | EXPR ;
                | let [mut] EXPR = EXPR ;
                | while EXPR BLOC
                | return [EXPR] ;
                | IF_EXPR
IF_EXPR      ⟶  if EXPR BLOC [ else (BLOC | IF_EXPR) ]
EXPR         ⟶  cste_ent | true | false
                | idf
                | EXPR BINAIRE EXPR
                | UNAIRE EXPR
                | EXPR . idf
                | EXPR . len ( )
                | EXPR [ EXPR ]
                | idf ( [ EXPR (, EXPR)* ] )
                | idf { [ idf : EXPR (, idf : EXPR)* ] }
                | Vec ! [ [ EXPR (, EXPR)* ] ]
                | print ! ( EXPR )
                | BLOC
                | ( EXPR )
BINAIRE      ⟶  + | - | * | / | && | || | < | <= | > | >= | == | !=
UNAIRE       ⟶  - | ! | * | &
```

FIGURE 1 – Définition de la grammaire de Mini-Rust

L'objectif était de rendre cette grammaire LL(1). Pour ce faire, nous avons dans un premier temps tenté de répartir les tâches avant de tous se mettre à travailler en équipe sur un même fichier. Après un premier échec, nous avons décidé de recommencer la grammaire en procédant à l'implémentation des règles les plus fondamentales de la spécification. Cette stratégie s'est avérée payante, et a conduit au fichier mini_rust.g dont le contenu est décrit en annexe.

Une difficulté que nous avons rencontré lors de la spécification de cette grammaire sous ANTLR a été la gestion des priorités des opérateurs. Afin de gérer la priorité des opérateurs, nous avons pris pour référence le tableau ci-dessous.

Opérateur	Associativité	Priorité
=	droite	faible
||	gauche	
&&	gauche	
== != < <= > >=	–	
+ -	gauche	
* /	gauche	
! *(unaire) -(unaire) &	–	
[]	–	
.	–	forte

FIGURE 2 – Tableau de la gestion des priorités

1.2 Conception Arbre Syntaxique Abstrait

L'analyse syntaxique nous fournit un arbre syntaxique (AST) qui est obtenu directement d'après les règles de la grammaire et l'analyse du code source.

Afin de créer l'AST, nous avons d'abord défini les différents noeuds dans la grammaire puis nous avons utilisé la structure de données Tree de ANTLR dans le partie java du projet. Chaque élément de l'AST que ce soit la racine, les noeuds ou les feuilles sont tous traités comme des arbres ayant ou non des parents et des enfants.
Nous avons défini une fonction *traverse* par noeuds définis dans la grammaire et que nous retrouvons dans la classe **TreeTraversal.java**. C'est dans ces fontions que les contrôles sémantiques et la table des symboles, que nous décrirons par la suite, sont appelés.

La représentation graphique de l'AST peut s'obtenir en lançant le mode débug de antlrwork.

Nous allons maintenant expliquer le fonctionnement de la construction de l'AST à l'aide d'exemples.

L'instruction "LET" (figure 3) permet d'attribuer une valeur non mutable (elle ne peut pas changer de valeur) à une variable. Sur l'arbre, le noeud LET possède deux enfants, la variable et la valeur. (Il est possible d'associer seulement une variable avec let sans lui attribuer de valeur mais nous considérer cela comme une exception dans la partie contrôle sémantique)

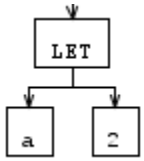

FIGURE 3 – Gestion de let

L'instruction "LETMUT" (figure **??**) fonctionne comme l'instruction "LET" (deux enfants, la variable et la valeur) hormis le fait qu'elle rende une variable mutable. Puisque le caractère mutable et non mutable est important dans le traitement des variables par la suite, nous avons donc fait un noeud différent pour les deux instructions.

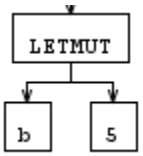

FIGURE 4 – Gestion de letmut

Les instructions "LET" et "LETMUT" (figure 5) sont utilisées lors d'opérations mathématiques, dans ce cas-là l'enfant de droite du noeud d'instruction qui correspond la valeur du résultat de l'opération, prend la valeur de l'opération mathématique ayant la priorité la moins grande.

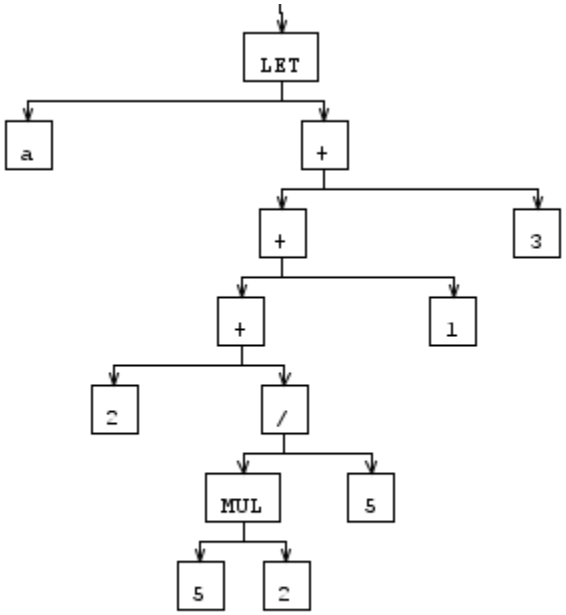

FIGURE 5 – Exemple d'utilisation de let

Le noeud "IF" (figure 6) a pour enfants, la condition du if et le bloc d'instructions à exécuter si la condition est validée.

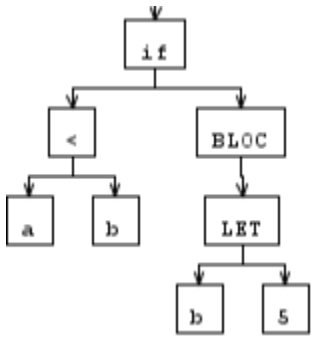

FIGURE 6 – Gestion de if

La valeur de l'enfant de gauche prend, dans le cas où il y a plusieurs conditions (avec les opérateurs and et or), la valeur de l'opérateur logique avec la priorité la plus faible. (figure 7)

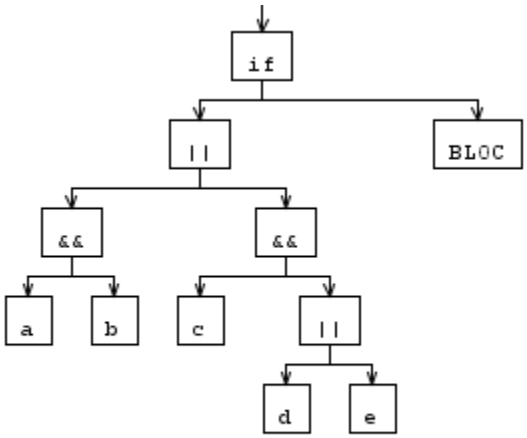

FIGURE 7 – Exemple d'utilisation d'un if avec plusieurs conditions

L'instruction while (figure 8) fonctionne comme if avec une condition et un bloc d'instructions en cas de condition validée. Nous avons donc dévoloppé le noeud bloc dans l'exemple suivant.

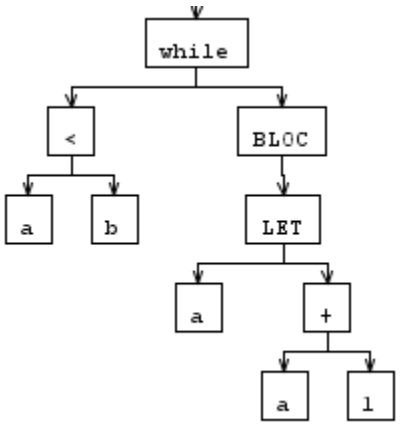

FIGURE 8 – Exemple d'utilisation de while

Dans l'exemple suivant (figure 9), nous avons montré comment notre arbre gère l'initialisation et l'attribution de valeur dans un tableau. Le nombre d'enfant attribué au noeud tableau dépend du nombre d'élément qu'il possède.

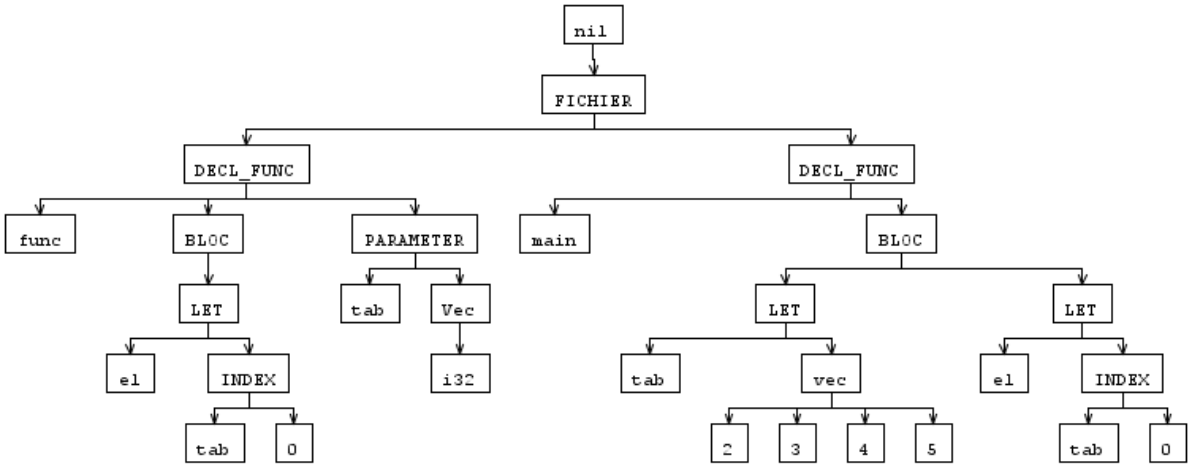

FIGURE 9 – Gestion des tableaux

Pour déclarer une fonction (figure 10), certains noeuds sont obligatoires : le nom de la fonction et son bloc d'instruction qui peut être vide. D'autres comme le type de retour et les paramètres ne le sont pas (à défaut, on considère respectivement que la fonction a un type de retour "void" ou que la fonction n'a pas d'argument). Les noeuds "DECL_FUNC" se construisent dans l'ordre d'apparition des fonctions.

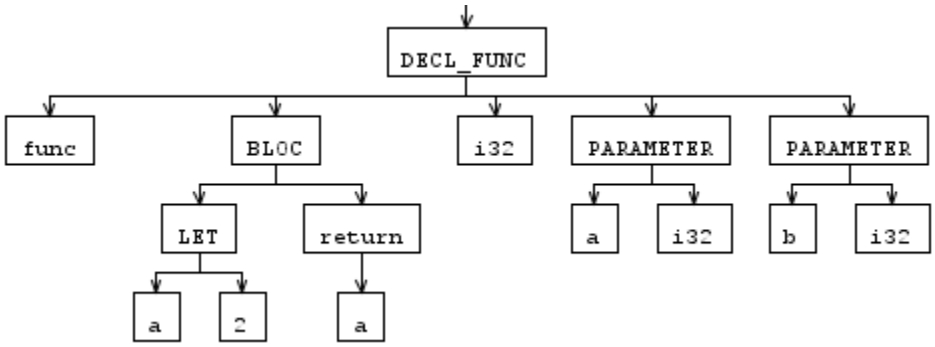

FIGURE 10 – Gestion de deux déclarations de fonctions

Dans le cas où la fonction a un type de retour et un ou des paramètres (figure 11), le feuille correspondant à la valeur du type apparaît avant les paramètres. Dans le cas où la fonction a plusieurs paramètres, les noeuds "PARAMETER" sont créés dans l'ordre d'apparition sur le programme à traiter.
Les paramètres sont constitués de deux noeuds, le noeud de gauche correspondat au nom du paramètre et celui de droite à son type.

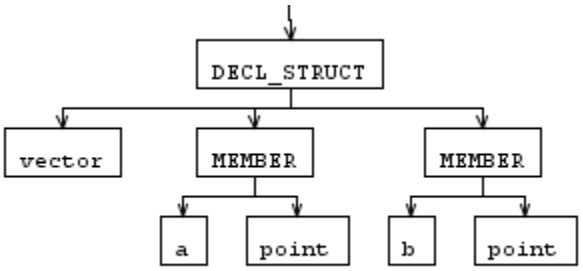

FIGURE 11 – Gestion des paramètres

L'appel à une fonction (figure 15) se fait à l'aide du noeud "FUNCTION_CALL". Dans le cas où la fonction n'a pas d'argument, ce noeud a un fils qui prend la valeur du nom de cette dernière.

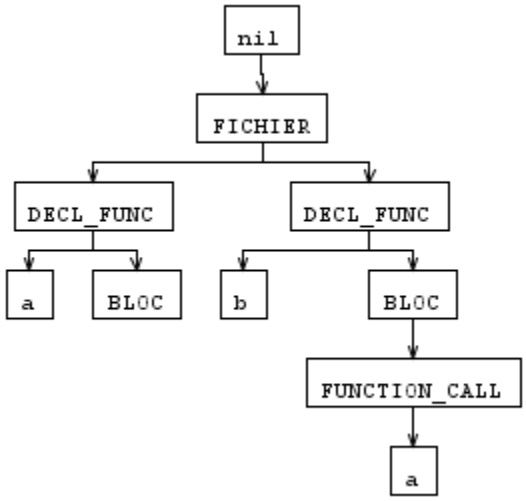

FIGURE 12 – Gestion de l'appel d'une fonction sans paramètre

Dans le cas où la fonction appelée a des paramètres (figure 13), les enfants du noeud "FUNCTION_CALL" seront dans l'ordre le nom de la fonction appelée et les différents arguments dans le même ordre de définition que les paramètres dont ils se réfèrent.

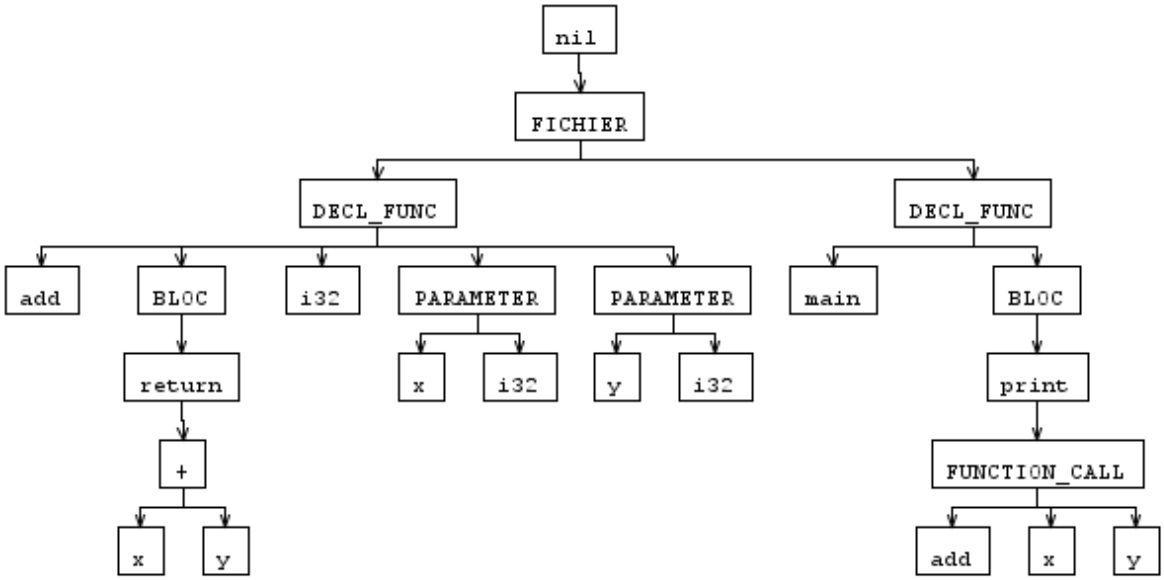

FIGURE 13 – Gestion d'appel de fonction avec paramètres

Dans le dernier exemples, nous allons montrer comment sont gérés par l'arbre la déclaration et l'appel de structure (figure 14).
Le noeud "DECL_STRUCT" possède un noeud correspond au nom de la structure et un nombre de noeuds équivalant à ses éléments (il est possible à cette étape de l'analyse de code de faire des structures vides, cependant nous l'avons traité comme étant une exception sémantique car inutilisable).

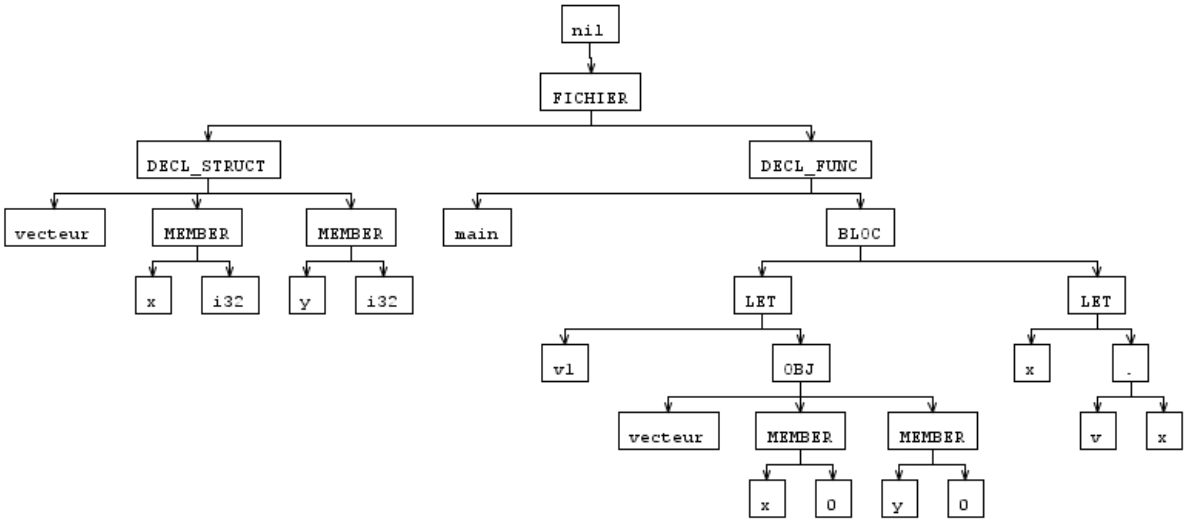

FIGURE 14 – Gestion des structures

2 Analyse Sémantique

2.1 Structure de la table des symboles

La construction de la table des symboles (TDS) est une étape essentielle qui a lieu lors de la compilation d'un programme. Dans la compilation, elle survient après l'étape de l'analyse lexicale et syntaxique. Elle sert pour l'analyse sémantique mais également pour la génération de code en conservant le déplacement des variables. Une entrée dans la TDS correspond à une déclaration d'objet dans le code. Chaque fonction, structure, bloc anonyme définissent un nouvel sous-espace de noms et donc une nouvelle TDS.

Afin d'implémenter la TDS, nous avons tout d'abord défini une classe pour chacun des trois symboles que nous avons identifié : variable, fonction, structure. Il nous a donc fallu réfléchir en parallèle à des structures de données adaptées à notre problème que nous pouvons retrouver ci-dessous.

Pour vérifier si la création de la table des symboles se fait correctement, nous avons imprimé, durant cette étape, les TDS une fois créées

— les **HashMap<String, Symbol>** qui permettent de récupérer un symbole à partir de son nom.
— les **HashMap<Integer, SymbolTable>** qui sont utilisés pour définir les blocs
— les **LinkedList<VariableSymbol>** qui permettent de stocker les paramètres des fonctions.
— les **Stack<>** qui permettent d'empiler les TDS afin de récupérer plus facilement leur niveau d'imbrication.

Pour chacun des trois symboles identifiés, nous stockons un certain nombre d'informations que l'on retrouve ci-dessous.

Symbole	Informations
Fonction	Nom, portée, type de retour, imbrication, décalage(offset), numéro de région
Structure	Nom, portée, imbrication, décalage(offset), numéro de région
Variable	Nom, type, portée, imbrication, décalage, région

```
|0|0|
|NAME                |OFFSET|SYMB          |TYPE          |
|vecteur             |0     |STRUCT(1)     |null
|main                |0     |FUNC(4)       |void
|addition_vecteurs   |0     |FUNC(3)       |vecteur
|data                |0     |STRUCT(2)     |null


|1|1|
|NAME                |OFFSET|SYMB          |TYPE          |
|x                   |2     |VAR           |i32
|y                   |4     |VAR           |i32


|4|1|
|NAME                |OFFSET|SYMB          |TYPE          |
|d1                  |2     |VAR           |data
|d2                  |7     |VAR           |data
|d3                  |12    |VAR           |data
|d4                  |17    |VAR           |data
|l                   |22    |VAR           |Vec<data>
|result              |27    |VAR           |vecteur


|3|1|
|NAME                |OFFSET|SYMB          |TYPE          |
|l                   |-4    |VAR           |Vec<data>
|i                   |2     |VAR           |i32
|result              |4     |VAR           |vecteur


|2|1|
|NAME                |OFFSET|SYMB          |TYPE          |
|v                   |2     |VAR           |vecteur
|to_add              |6     |VAR           |bool
```

FIGURE 15 – TDS crée à partir de l'exemple 4

2.2 Liste des contrôles sémantiques implémentés

Nous avons mis en place un système d'exceptions afin de vérifier la sémantique du code. L'AST est parcouru et teste chacune des instructions. Chacune des exceptions est définie dans une classe *Java* qui lui est associée.

Pour organiser les classes d'exception nous avons fait de l'héritages par type. Toutes les classes d'exception, sauf **UnknownNodeException** qui n'est pas un contrôle sémantique, héritent de la classe abstraite **SemanticException**.

Le tableau ci-dessous présente la liste des contrôles sémantiques implémentés par chacun des membres du groupe de projet.

Membre	Contrôle Sémantique
Laetitia Huret (7)	- **NoMainFoundException** : Vérifie que le fichier possède une fonction main - **RedefiningStructureException** : Vérifie que la structure définie est unique - **RedefiningStructureElementException** : Vérifie que les éléments d'une structure sont uniques - **LenNotAtEndException** : Vérifie que le point **.len** est bien utilisé à la fin *len* - **UsingIndexAccessorOnNotVecException** : Vérifie que l'index n'est utilisé que sur un vecteur -**PrintUndefinedSymbolException** : Vérifie que la fonction *print* fait appel soit à un entier soit à un symbole défini dans la TDS
Maylis Willaime–Angonin (7)	- **EmptyVectorException** : Vérifie qu'un tableau n'est pas vide - **AndOrWithoutBooleanException** : Vérifie qu'une expression avec des opérateurs logiques and et or sont booléennes - **OperationWithNoIntException** : Vérifie qu'une opération mathématique n'est effectuée qu'entre des éléments de type i32 - **IsNotWithoutBoolException** : Vérifie que l'opérateur**!** est devant une expression booléenne - **UndefinedCalledElementException** : Vérifie qu'un élément appelé a bien été défini - **UnknownTypeException** : Vérifie que le type appelé existe dans le langage - **EmptyFileException** : Vérifie que le fichier lancé n'est pas vide - **RedefiningFunctionException** : Vérifie que la fonction est unique

Najib Aghenda (8)	- **WrongTypeReturnException** : Vérifie que le type de l'élément retourné par une fonction est du type que doit retourner la foncton - **DifferentTypeException** : Vérifie que les éléments dans un vecteur sont du même type - **WrongNbArgumentException** : Vérifie que la fonction est appelée avec le bon nombre d'arguments - **WhileWithoutBoolException** : Vérifie que l'expression évaluée dans une instruction *while* est bien un booléen - **IfWithoutBoolException** : Vérifie que l'expression évalue dans une instruction *if* est bien un booléen - **UndefinedSymbolException** : Vérifie qu'un symbole est bien défini -**MainWithArgumentException** : Vérifie que la fonction **main** ne possède aucune argument. - **RedefiningStructureElementException** : Vérifie que la structure est unique
Damien Vantourout (9)	- **UnusuableVariableException** : Vérifie que la variable créée par un let n'est pas nulle - **NonMutableException** : Vérifie qu'une variable non mutable ne change pas de valeur - **RedefiningVariableTypeException** : Vérifie que la variable ne change pas de type suite à une nouvelle attribution de valeur - **WrongNumberCalledElementException** : Vérifie que lors de l'appel d'une structure le nombre d'éléments qu'elle possède soit le même que le nombre d'éléments dans sa déclaration - **WrongTypeCalledElementException** : Vérifie que lors de l'appel d'une structure les types des éléments de son appel et de sa déclaration soient les mêmes - **UndefinedSymbolException** : Vérifie que le symbole appelé est déjà présent dans la TDS - **PrintVoidException** : Vérifie que la variable appelée n'est pas nulle - **OperationWithNoIntException** : Vérifie que les opérations arithmétiques se fait avec deux entiers -**WrongTypeArgumentException** : Vérifie que les arguments d'une fonction sont du bon type

Nous avons choisi de faire l'exception **UnusuableVariableException** car si une valeur est non mutable et nulle, elle sera inutilisable. De même, puisque dans le langage que nous avons implémenté, il n'est pas possible d'ajouter de valeurs à un tableau, nous avons décidé de faire une exception sémantique sur les tableaux vides qui sont inutilisables.

3 Génération du code assembleur

Une fois la TDS et les contôles sémantiques implémentés, nous avons dû nous pencher sur la génération de code. Ce code assembleur est généré au format microPiup, la documentation étant bien fournie, nous avons pu réaliser rapidement ensemble quelques fonctions basiques, telles qu'une addition de deux opérandes, afin de bien comprendre les bases de la génération de code pour pouvoir se répartir plus simplement les tâches de la génération de code après.
La façon dont nous avons écrit le code en assembleur est décrit en annexe dans la partie "Schémas de traduction". Ce code est ensuite généré par la fonction **compile()** en *Java*. Nous avons fait en sorte que l'évolution des états de la pile soient affichés après avoir compilé un fichier.

4 Conclusion

Ce projet a été pour nous l'occasion de mettre en application les notions abordées en cours de *Traduction I*. La durée du projet ainsi que son jalonnement nous a permis de bien assimiler les différentes étapes qui entrent en lieu lors de la création du compilateur pour le langage Mini-Rust.

Tout au long du projet, nous avons dû faire face à des difficultés liées au sujet mais également aux dates limites imposées régulièrement. Ces difficultés nous ont permis de nous améliorer afin de fournir un rendu de meilleur qualité. Ce projet nous a fait prendre conscience de la réelle difficulté que constitue la création d'un compilateur et de la rigueur que cette tâche demande.

Malgré un projet qui répond à de nombreuses exigences, certaines améliorations au code, comme par exemple l'ajout de contrôles sémantiques ou encore la gestion des pointeurs auraient été possibles mais nous n'avons pas poursuivi par faute de temps et par priorité des autres tâches.

Pour conclure, nous sommes globalement satisfaits du compilateur que nous présentons. Le projet rendu fait fonctionner de nombreux contrôles sémantiques et génère le code assembleur efficacement.

A Grammaire

A.1 mini_rust.g

```
1  grammar mini_rust;
2
3  options {
4   language = Java;
5   k = 1;
6   output = AST;
7  }
8
9  tokens{
10  FICHIER;
11  DECL_FUNC;
12  DECL_STRUCT;
13  TYPE;
14  PARAMETER;
15  BLOC;
16  INDEX;
17  FUNCTION_CALL;
18  UNARY_MINUS;
```

```
19    OBJ;
20    MEMBER;
21    LETMUT;
22    MUL;
23    NEG;
24    POINTER;
25    REF;
26
27    LPAREN = '(';
28    RPAREN = ')';
29    LBRACKET = '{';
30    RBRACKET = '}';
31    LSQBRACKET = '[';
32    RSQBRACKET = ']';
33
34    GT = '>';
35    GE = '>=';
36    LT = '<';
37    LE = '<=';
38    EQ = '==';
39    NE = '!=';
40    AND = '&&';
41    OR = '||';
42
43    PLUS = '+';
44    MINUS = '-';
45    DIV = '/';
46    STAR = '*';
47    EXCL = '!';
48
49    ASSIGN = '=';
50    DOT = '.';
51    AMPS = '&';
52    LEN = 'len';
53
54    LET = 'let';
55    MUT = 'mut';
56    FN = 'fn';
57    STRUCT = 'struct';
58    WHILE = 'while';
59    IF = 'if';
60    ELSE = 'else';
61    RETURN = 'return';
62    VEC_MACRO = 'vec';
63    PRINT_MACRO = 'print';
64
65    VEC_TYPE = 'Vec';
66    INT32_TYPE = 'int32';
67    BOOL_TYPE = 'bool';
68    TRUE = 'true';
69    FALSE = 'false';
70
71    COMMA = ',';
72    SEMICOLON = ';';
73    COLON = ':';
74    ARROW = '->';
75  }
76
77  @header {
```

```
78   package grammar;
79  }
80
81  @lexer::header {
82   package grammar;
83  }
84
85  fichier
86   :
87   (decl)* -> ^(FICHIER decl*)
88   ;
89
90  decl
91   :
92     decl_func
93   | decl_struct
94   ;
95
96  decl_func
97   :
98   FN IDF LPAREN (parameter (COMMA parameter)*)? RPAREN (ARROW type)? bloc
99   -> ^(DECL_FUNC IDF bloc (type)? (parameter)*)
100  ;
101
102 decl_struct
103  :
104  STRUCT idf=IDF LBRACKET (i+=IDF COLON t+=type (COMMA i+=IDF COLON t+=type)*)? RBRACKET
105  -> ^(DECL_STRUCT $idf ^(MEMBER $i $t)*)
106  ;
107
108 type
109  :
110    IDF
111  | VEC_TYPE LT type GT -> ^(VEC_TYPE type)
112  | AMPS type -> ^(REF type)
113  | INT32_TYPE
114  | BOOL_TYPE
115  ;
116
117 parameter
118  :
119  IDF COLON type -> ^(PARAMETER IDF type)
120  ;
121
122 bloc
123  :
124  LBRACKET instruction_bloc RBRACKET -> ^(BLOC instruction_bloc?)
125  ;
126
127 instruction_bloc
128  :
129    instruction instruction_bloc
130  | (expr)? (SEMICOLON instruction_bloc)? -> expr? instruction_bloc?
131  ;
132
133 instruction
134  :
135    LET instruction_let SEMICOLON -> instruction_let
136  | WHILE expr bloc -> ^(WHILE expr bloc)
```

```
137   | RETURN (expr)? SEMICOLON -> ^(RETURN expr?)
138   | if_expr
139   ;
140
141 instruction_let
142   :
143     MUT expr (ASSIGN structure_initialisation_expr)?
144     -> ^(LETMUT expr (structure_initialisation_expr)?)
145   | expr (ASSIGN structure_initialisation_expr)?
146     -> ^(LET expr (structure_initialisation_expr)?)
147   ;
148
149 structure_initialisation
150   :
151  LBRACKET (i+=IDF COLON o+=structure_initialisation_expr (COMMA i+=IDF COLON
152  o+=structure_initialisation_expr)*)? RBRACKET -> ^(MEMBER $i $o)*
153   ;
154
155 structure_initialisation_expr
156   :
157     expr
158     (
159        -> expr
160      | structure_initialisation -> ^(OBJ expr structure_initialisation)
161     )
162   ;
163
164 if_expr
165   :
166  IF expr bloc (else_expr)? -> ^(IF expr bloc (else_expr)?)
167   ;
168
169 else_expr
170   :
171  ELSE (bloc -> ^(ELSE bloc) | if_expr -> ^(ELSE if_expr))
172   ;
173
174 expr
175   :
176     expr_ou
177   | bloc
178   ;
179
180 dot_factorisation
181   :
182     IDF -> IDF
183   | LEN LPAREN RPAREN -> LEN
184   ;
185
186 expr_ou
187   :
188  (e1=expr_et -> $e1) (OR e2=expr_et -> ^(OR $expr_ou $e2))*
189   ;
190
191 expr_et
192   :
193  (e1=expr_comp -> $e1) (AND e2=expr_comp -> ^(AND $expr_et $e2))*
194   ;
195
```

```
196 expr_comp
197   :
198   (e1 = expr_plus -> $e1)
199   (
200       LT e2=expr_plus -> ^(LT $expr_comp $e2)
201     | LE e2=expr_plus -> ^(LE $expr_comp $e2)
202     | GT e2=expr_plus -> ^(GT $expr_comp $e2)
203     | GE e2=expr_plus -> ^(GE $expr_comp $e2)
204     | EQ e2=expr_plus  -> ^(EQ $expr_comp $e2)
205     | NE e2=expr_plus  -> ^(NE $expr_comp $e2)
206   )*
207   ;
208
209 expr_plus
210   :
211   (e1=expr_mult -> $e1)
212   (
213         PLUS e2=expr_mult -> ^(PLUS $expr_plus $e2)
214       | MINUS e2=expr_mult -> ^(MINUS $expr_plus $e2)
215   )*
216   ;
217
218 expr_mult
219   :
220   (e1=expr_unaire -> $e1)
221   (
222       STAR e2=expr_unaire -> ^(MUL $expr_mult $e2)
223     | DIV e2=expr_unaire -> ^(DIV $expr_mult $e2)
224   )*
225   ;
226
227 expr_unaire
228   :
229     MINUS expr_unaire -> ^(UNARY_MINUS expr_unaire)
230   | (
231         EXCL expr_unaire -> ^(NEG expr_unaire)
232       | STAR expr_unaire -> ^(POINTER expr_unaire)
233       | AMPS expr_unaire -> ^(REF expr_unaire)
234     )
235   | (a=atom -> $a)
236     (
237         LSQBRACKET expr RSQBRACKET -> ^(INDEX $expr_unaire expr)
238       | DOT dot_factorisation -> ^(DOT $expr_unaire dot_factorisation)
239     )*
240   ;
241
242 atom
243   :
244     CSTE_ENT
245   | TRUE
246   | FALSE
247   | IDF
248       (
249             -> IDF
250           | LPAREN (expr (COMMA expr)*)? RPAREN -> ^(FUNCTION_CALL IDF expr*)
251       )
252   | LPAREN expr RPAREN -> expr
253   | VEC_MACRO EXCL LSQBRACKET (e+=expr (COMMA e+=expr)*)? RSQBRACKET
254     -> ^(VEC_MACRO ($e)*)
```

```
255   | PRINT_MACRO EXCL LPAREN expr RPAREN -> ^(PRINT_MACRO expr)
256   ;
257
258 IDF : (LOWERCASE | UPPERCASE | '_') (LOWERCASE | UPPERCASE | DIGIT | '_')* ;
259 CSTE_ENT : DIGIT+ ;
260 COMMENTS : '/*' .* '*/' { $channel = HIDDEN ; } ;
261 WS : (' ' | '\r' | '\n' | '\t') { $channel = HIDDEN ; } ;
262
263 fragment LOWERCASE : 'a'..'z';
264 fragment UPPERCASE : 'A'..'Z';
265 fragment DIGIT : '0'..'9';
```

B Schémas de Traduction

Ce code assembleur est le code qu'on retrouve dans tout les fichiers, il définit des alias pour les registres usuels, l'adresse de base de la pile et le point de départ du programme

```
1  // Registres usuels
2  SP EQU R15
3  WR EQU R14
4  BP EQU R13
5  // Trappes usuelles
6  EXIT_EXC EQU 64
7  READ_EXC EQU 65
8  WRITE_EXC EQU 66
9  // Definitions de la valeur NULL
10 NUL EQU 0
11 NULL EQU 0
12 NIL EQU 0
13 // Adresse debut de la pile
14 STACK_ADDRS EQU 0x1000
15 // Adresse debut de programme
16 LOAD_ADDRS EQU 0xFE00
17 ORG LOAD_ADDRS
18 START main_
19 LDW SP, #STACK_ADDRS
20 LDW BP, #NIL
```

Code assembleur de la fonction print, elle affiche à l'écran une chaine de caractères

```
1  print_
2  // Ouverture de l'environnement
3  STW BP, -(SP)
4  LDW BP, SP
5  // Reservation sur la pile des variables
6  LDQ 0, R0
7  SUB SP, R0, SP
8  LDW R0, (BP)4
9  TRP #WRITE_EXC
10 // Fermeture de l'environnement
11 LDW SP, BP
12 LDW BP, (SP)+
13 RTS
```

Code assembleur de la fonction printi, elle affiche à l'écran un chiffre (convertit en chaine de caractères avec la fonction itoa avant affichage)

```
1
2  // Ouverture de l'environnement
3  STW BP, -(SP)
4  LDW BP, SP
5  // Reservation sur la pile des variables
6  LDQ 0, R0
7  SUB SP, R0, SP
8  // r serve 7+1 = 8 caract res en pile
9  // (entier pair sup rieur    7 demand pour pas d saligner pile)
10 ADI SP, SP, #-8
11 ADI SP, SP, #-2
12 LDW R0, (BP)4
```

```
13 stw r0, (BP)-10    // sauve r0    l'adresse BP-10
14 // itoa(value, text, 10);
15 // appelle itoa avec i = value, p = text, b = 10
16 ldw r0, #10        // charge 10 (pour base d cimale) dans r0
17 stw r0, -(SP)      // empile contenu de r0 (param tre b)
18 adi BP, r0, #-8    // r0 = BP - 8 = adresse du tableau text
19 stw r0, -(SP)      // empile contenu de r0 (param tre p)
20 ldw r0, (BP)-10    // charge r0 avec value
21 stw r0, -(SP)      // empile contenu de r0 (param tre i)
22 jsr @itoa_         // appelle fonction itoa d'adresse itoa_
23 adi SP, SP, #6     // nettoie la pile des param tres
24 // de taille totale 6 octets
25 // print(text);
26 adi BP, r0, #-8    // r0 = BP - 8 = adresse du tableau text
27 stw r0, -(SP)     // empile contenu de r0 (param tre p)
28 jsr @print_        // appelle fonction print d'adresse print_
29 adi SP, SP, #2     // nettoie la pile des param tres
30      // de taille totale 2 octets
31 // Fermeture de l'environnement
32 LDW SP, BP
33 LDW BP, (SP)+
34 RTS
```

Code pour assembleur pour l'affectation d'une variable

```
1 fn main() {
2     let a = 2;
3     let b = 5;
4
5     let c = a + b;
6 }
```

```
1  main_
2  // Ouverture de l'environnement
3  STW BP, -(SP)
4  LDW BP, SP
5  // Reservation sur la pile des variables
6  LDQ 6, R0
7  SUB SP, R0, SP
8  // let a = 2
9  LDW R0, #2
10 STW R0, (BP)-2
11 // let b = 5
12 LDW R0, #5
13 STW R0, (BP)-4
14 // let c = a + b
15 LDW R0, (BP)-2
16 LDW R1, (BP)-4
17 ADD R0, R1, R0
18 STW R0, (BP)-6
19 // Fermeture de l'environnement
20 LDW SP, BP
21 LDW BP, (SP)+
22 TRP #EXIT_EXC
23 JEA @main_
```

Code assembleur pour le changement de valeur d'une variable (mut) et d'une boucle while

```
fn main() {
    let a = 10;
    let mut i = 0;

    while i < a {
        let i = i + 1;
    }
}
```

```
main_
// Ouverture de l'environnement
STW BP, -(SP)
LDW BP, SP
// Reservation sur la pile des variables
LDQ 4, R0
SUB SP, R0, SP
// let a = 10
LDW R0, #10
STW R0, (BP)-2
// let i = 0
LDW R0, #0
STW R0, (BP)-4
// while
bwhile_868693306
// i < a
LDW R0, (BP)-4
LDW R1, (BP)-2
CMP R0, R1
BGE ewhile_868693306-$-2
// let i = i + 1
LDW R0, (BP)-4
LDW R1, #1
ADD R0, R1, R0
STW R0, (BP)-4
BMP bwhile_868693306-$-2
ewhile_868693306
// Fermeture de l'environnement
LDW SP, BP
LDW BP, (SP)+
TRP #EXIT_EXC
JEA @main_
```

Code assembleur pour une fonction avec une condition comportant un et. Pour une condition composée uniquement de et, il suffit d'enchaîner les CMP et de faire un saut BCC à la fin du bloc dès qu'une condition n'est pas respectée.

```
fn main() {
    let mut i = 5;

    while i > 0 && i < 10 {
        print!(i);
        let i = i + 1;
```

```
7       }
8   }
```

```
1   main_
2   // Ouverture de l'environnement
3   STW BP, -(SP)
4   LDW BP, SP
5   // Reservation sur la pile des variables
6   LDQ 2, R0
7   SUB SP, R0, SP
8   // let mut i = 5
9   LDW R0, #5
10  STW R0, (BP)-2
11  // while
12  bwhile_868693306
13  // i > 0
14  LDW R0, (BP)-2
15  LDW R1, #0
16  CMP R0, R1
17  BLE ewhile_868693306-$-2
18  // && i < 10
19  LDW R0, (BP)-2
20  LDW R1, #10
21  CMP R0, R1
22  BGE ewhile_868693306-$-2
23  // print
24  LDW R0, (BP)-2
25  STW R0, -(SP)
26  JSR @printi_
27  // let i = i + 1
28  LDW R0, (BP)-2
29  LDW R1, #1
30  ADD R0, R1, R0
31  STW R0, (BP)-2
32  BMP bwhile_868693306-$-2
33  ewhile_868693306
34  // Fermeture de l'environnement
35  LDW SP, BP
36  LDW BP, (SP)+
37  TRP #EXIT_EXC
38  JEA @main_
```

Les conditions ou n'ont pas encore été implémentation pour la génération, mais on pourrait l'implémenter en générant le code assembleur suivant :

```
1   fn main() {
2       let mut i = 0;
3
4       while i == 0 || i == 1 {
5           print!(i);
6           let i = i + 1;
7       }
8   }
```

```
main_
// Ouverture de l'environnement
STW BP, -(SP)
LDW BP, SP
// Reservation sur la pile des variables
LDQ 2, R0
SUB SP, R0, SP
// let mut i = 0
LDW R0, #0
STW R0, (BP)-2
// while
bwhile_868693306
// i == 0
LDW R0, (BP)-2
LDW R1, #0
CMP R0, R1
BEQ econdwhile_868693306-$-2
// || i == 1
LDW R0, (BP)-2
LDW R1, #1
CMP R0, R1
BEQ econdwhile_868693306-$-2
BMP ewhile_868693306-$-2
econdwhile_868693306
// print
LDW R0, (BP)-2
STW R0, -(SP)
JSR @printi_
// let i = i + 1
LDW R0, (BP)-2
LDW R1, #1
ADD R0, R1, R0
STW R0, (BP)-2
BMP bwhile_868693306-$-2
ewhile_868693306
// Fermeture de l'environnement
LDW SP, BP
LDW BP, (SP)+
TRP #EXIT_EXC
JEA @main_
```

```
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

fn main() {
    int res = add(2, 5);
    print!(res);
}
```

Appel de fonctions

```
add_
// Ouverture de l'environnement
STW BP, -(SP)
LDW BP, SP
```

```
5  // return a + b
6  LDW R0, (BP)4
7  LDW R1, (BP)6
8  ADD R0, R1, R0
9  // Fermeture de l'environnement
10 LDW SP, BP
11 LDW BP, (SP)+
12 RTS
13
14 main_
15 // Ouverture de l'environnement
16 STW BP, -(SP)
17 LDW BP, SP
18 // Reservation sur la pile des variables
19 LDQ 2, R0
20 SUB SP, R0, SP
21 // add(2, 5)
22 LDW R0, #2
23 STW R0, -(SP)
24 LDW R0, #5
25 STW R0, -(SP)
26 JSR @add_
27 // res = add(2, 5)
28 STW R0, (BP)-2
29 // print!(res)
30 LDW R0, (BP)-2
31 STW R0, -(SP)
32 JSR @printi_
33
34 // Fermeture de l'environnement
35 LDW SP, BP
36 LDW BP, (SP)+
37 TRP #EXIT_EXC
38 JEA @main_
```

C Fichiers de tests

Nous avons tenté, tout au long du projet, d'implémenter des programmes afin de tester les différentes étapes de notre compilateur. Ci-dessous, nous trouvons des programmes de tests :

Ce premier test permet de tester la déclaration d'une fonction, son appel, le fonctionnement d'un bloc *if* et du *return*. Ce test est disponible dans le fichier **maximum.rs**.

```
1
2  fn maximum(x : i32, y : i32) -> i32 {
3   if x > y {
4    return x;
5   }
6   else {
7    return y;
8   }
9  }
10
11 fn main(){
12  maximum(1,2);
13 }
14
15 }
```

Ce test est disponible dans le fichier **occurence.rs** et permet de tester la déclaration d'une variable avec *let*, la boucle *while*, les tableaux mais également le vecteur et le print.

```
1  fn first_occ(chaine : Vec<idf>, car : idf) -> i32 {
2   let i = 0;
3   while chaine[i]!='\0' {
4    if chaine[i] == car{
5    let i = i +1;
6    return i;
7    }
8    let i = i + 1;
9   }
10  return 0;
11 }
12
13 fn last_occ(chaine : Vec<idf>, car : idf) -> i32 {
14  let i = 0;
15  let retenu = 0;
16  while (chaine[i]!='\0'){
17   if chaine[i] == car {
18    let retenu = i;
19   }
20   let i = i + 1;
21  }
22  let retenu = retenu + 1;
23  return retenu;
24 }
25
26 fn main(){
27  let chaine = Vec![t, e, s, t];
28  print!(last_occ(chaine, t));
29  print!(first_occ(chaine, s));
30 }
```

Ce test est disponible dans le fichier **symetrie_points.rs** et permet de tester la déclaration et l'initialisation d'une structure.

```
1  struct Point{
2   id : i32,
3   x : i32,
4   y : i32
5  }
6
7  fn affiche_id(p : Point){
8   print!(p.id);
9  }
10
11 fn affiche_x(p : Point){
12  print!(p.x);
13 }
14
15 fn affiche_y(p: Point){
16  print!(p.y);
17 }
18 fn symetrique(p : Point, choix: i32) -> i32 {
19  if choix == 0 {
20   let x_2 = p.x;
21   let y_2 = -p.y;
22   let p.x = x_2;
23   let p.y = y_2;
24   affiche_x(p);
25   affiche_y(p);
26   return 1;
27  }
28  else if choix == 1 {
29   let x_2 = -p.x;
30   let y_2 = p.y;
31   let p.x = x_2;
32   let p.y = y_2;
33   affiche_x(p);
34   affiche_y(p);
35   return 1;
36  }
37  else if choix == 2 {
38   let x_2 = -p.x;
39   let y_2 = -p.y;
40   let p.x = x_2;
41   let p.y = y_2;
42   affiche_x(p);
43   affiche_y(p);
44   return 1;
45  }
46  else {
47   return 0;
48  }
49 }
50
51
52 fn main(){
53  let A  = Point {id : "12",x : 4,y : 3};
54  let B = Point {id : 15, x : 9, y : -3};
55  affiche_id(A);
```

```
56    affiche_id(B);
57    symetrique(A,0);
58    symetrique(B,1);
59    symetrique(A,2);
60
61  }
```

D Contribution des Membres

	Najib Aghenda	Laetitia Huret	Damien Vantourout	Maylis Willaime
Grammaire	18h	15h	20h	17h
Arbre Syntaxique	16h	15h	25h	16h
Table des Symboles	14h	13h	15h	18h
Contrôles sémantiques	8h	6h	7h	8h
Génération de code ASM	10h	4h	10h	9h
Rapport	4h	1h	1h	2h
Total	70h	54h	77h	70h