



# Rapport d'activité 4

## Rapport d'activité 4 PCL2

Professeurs référents : S. COLLIN et S. DA SILVA

TELECOM NANCY  
Projet de Compilation  
Deuxième année (S8)

Semaine 15  
Dernière mise à jour : le 12 avril 2023

### Membres du groupe :

Nicolas FRACHE  
Théo GOUREAU  
Cyrielle LACRAMPE-DITER  
Rida MOUSSAOUI

## Table des matières

<b>1</b>	<b>Gestion de projet</b>	<b>2</b>
1.1	Organisation générale . . . . .	2
1.2	Organisation du temps . . . . .	2
1.3	Missions des membres et temps estimé . . . . .	3
1.3.1	Nicolas FRACHE . . . . .	3
1.3.2	Théo GOUREAU . . . . .	4
1.3.3	Cyrielle LACRAMPE-DITER . . . . .	4
1.3.4	Rida MOUSSAOUI . . . . .	4
<b>2</b>	<b>Quelques schémas de traduction</b>	<b>4</b>
2.1	Liste des fonctionnalités implémentés . . . . .	4
2.2	Choix de conception . . . . .	5
2.2.1	Fonctions de base et macros en ARM64 . . . . .	5
2.2.2	Deuxième passe pour un AST optimisé . . . . .	6
2.3	Traduction du compilateurs des structures du langage . . . . .	8
2.3.1	Nouveau scope, appel de fonction et récursivité . . . . .	8
2.3.2	Structures conditionnelles imbriquées . . . . .	9
2.3.3	Enregistrements . . . . .	9
2.3.4	Tableaux . . . . .	9
<b>3</b>	<b>Annexe – Un exemple de programme : Fibonacci</b>	<b>10</b>
3.1	Code tiger . . . . .	10
3.2	Exemple de résultat . . . . .	10
3.3	AST de la fonction . . . . .	10
3.4	Extrait de code tiger . . . . .	11

# 1 Gestion de projet

## 1.1 Organisation générale

**Réunion** Environ une réunion était organisée tous les dix jours. Elle permettait de faire le point sur l'avancement et de mettre en commun les idées.

**Liste de tâches** Les comptes rendus étaient sous forme d'un tableur récapitulant les tâches et des détails sur comment les réaliser (comme des normes sur lesquelles nous nous mettions d'accord pendant ces réunions).

**Communication et outils** Nous communiquions sur Discord et utilisions un Drive pour partager les documents.

## 1.2 Organisation du temps

Cette période s'étend du mercredi 1 février (semaine 5) au jeudi 13 avril (semaine 15).

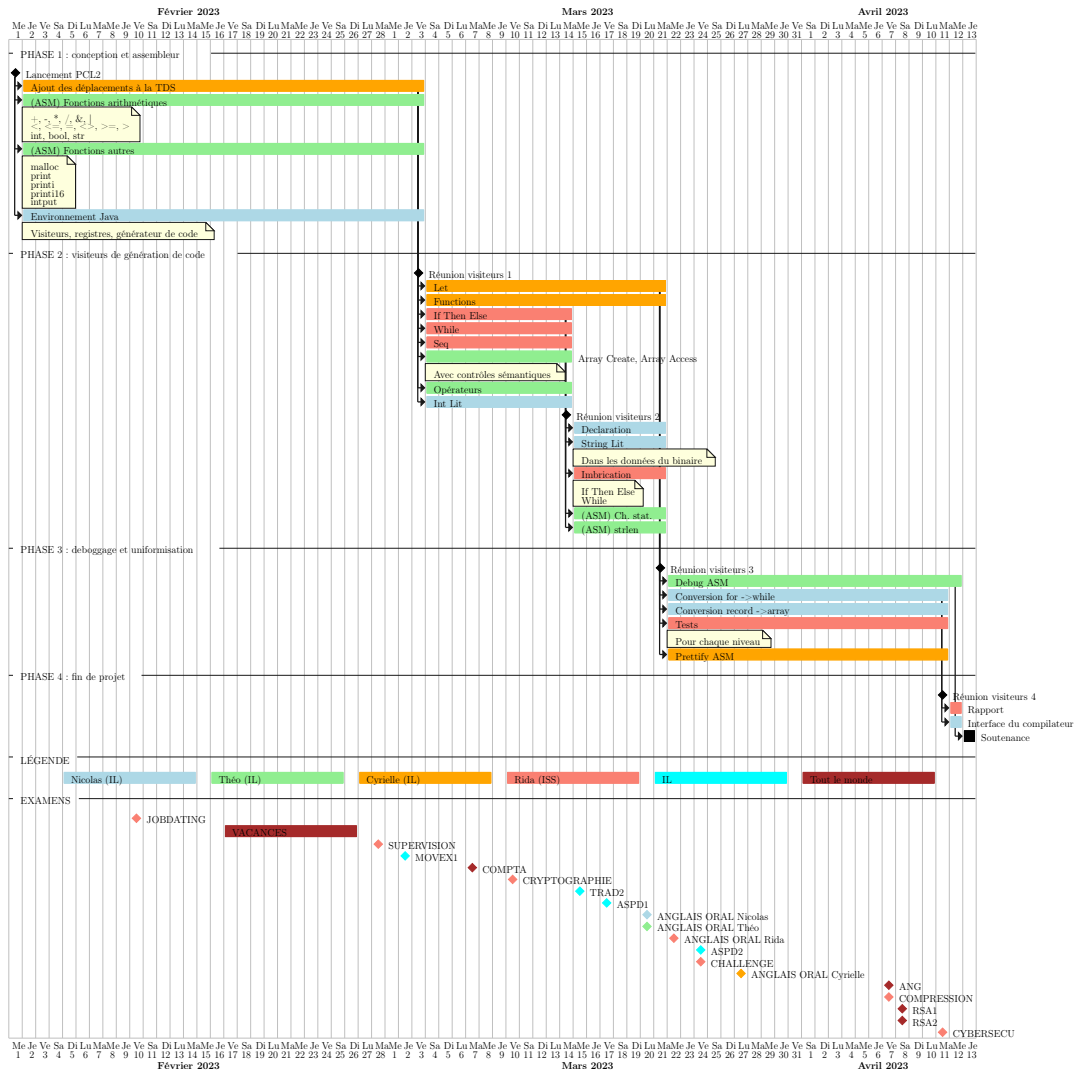


FIGURE 1 – Gantt simplifié de cette période

## 1.3 Missions des membres et temps estimé

### 1.3.1 Nicolas FRACHE

- Tests unitaires automatiques sur les contrôles sémantique (10 heures)
- Environnement Java pour les visiteurs (10 heures)
- Entiers, chaînes de caractères, Déclaration d'une variable (13 heures)
- Deuxième passe pour l'AST optimisé (17 heures)
- Débogage et interface (10 heures)

- Rédaction du rapport (3 heures)
- Réunions (7 heures)

Temps total estimé : 70 heures

### **1.3.2 Théo GOUREAU**

- Fonctions de base et macros en ARM64 (10 heures)
- Création et accès aux tableaux (3 heures)
- Opérateurs logiques et arithmétiques (15 heures)
- Débogage de l'ASM (30 heures)
- Rédaction du rapport (3 heures)
- Réunions (7 heures)

Temps total estimé : 68 heures.

### **1.3.3 Cyrielle LACRAMPE-DITER**

- Ajout de déplacement dans la TDS (10 heures)
- Let, appel des fonctions (25 heures)
- Rédaction du rapport (1 heure)
- Réunions (7 heures)

Temps total estimé : 43 heures

### **1.3.4 Rida MOUSSAOUI**

- Séquence, IfThenElse, while et gestion des imbrications (15 heures)
- Préparation des tests pour chaque niveau d'évaluation (15 heures)
- Rédaction du rapport (10 heures)
- Réunions (7 heures)

Temps total estimé : 47 heures

## **2 Quelques schémas de traduction**

### **2.1 Liste des fonctionnalités implémentés**

Nous avons implémenté les fonctionnalités :

1. Niveau 1 : déclaration des variables avec initialisation
2. Niveau 1 : affectations simples et affectations d'expressions arithmétiques
3. Niveau 1 : expressions arithmétiques avec variables et gestion de priorité
4. Niveau 1 : structures de contrôle : if, while, for non imbriquées

5. Niveau 2 : définition et appels des fonctions avec des arguments non récursifs
6. Niveau 3 : appels récursifs des fonctions
7. Niveau 3 : structures de contrôle : if, while, for imbriquées
8. Niveau 4 : l'opération print
9. Niveau 4 : implémentation des enregistrements(records) et tableaux
10. Niveau 4 : implémentation de break
11. Niveau 4 : la définition des types
12. Niveau 4 : fonctionnalités supplémentaires : *printi(entier)* qui affiche un entier sur la sortie standard, *intput()* qui lit un entier sur l'entrée standard

## 2.2 Choix de conception

### 2.2.1 Fonctions de base et macros en ARM64

Pour une organisation efficace du code, on a choisi d'implémenter des macros et des fonctions de base utilisés fréquemment dans le code assembleur.

- L'ensemble de macros dans le fichier *base\_macros.s*. Parmi ces macros on retrouve ceux qui servent à empiler ou déplier un ou plusieurs registres ainsi que les appels systèmes.
- L'ensemble de fonctions qui prennent des arguments pour la lecture ou de l'écriture sur l'entrée standard ou qui allouent de la mémoire pour sauvegarder des valeurs. Ces dernières sont localisées dans *base\_functions.s*
- L'ensemble de fonctions qui gèrent l'affectation et l'accès aux tableaux se trouvent dans *data\_functions.s*
- L'ensemble des fonctions qui gèrent les opérations arithmétiques (+, -, \*, /) et les opérateurs de comparaison (<, ≤, >, ≥, <>, =) sont dans *arithmetic\_functions.s*

### 2.2.2 Deuxième passe pour un AST optimisé

Nous avons fait le choix de faire passer notre AST généré par un troisième visiteur qui lui apportera des modifications de différents types.

**Traitement des boucles for** Afin de simplifier la génération de code, nous avons décidé de transformer le noeud *for* de l'AST en noeud *let* avec une boucle *while* correctement déclarée. La transformation est visible sur les figures 2 et 3.

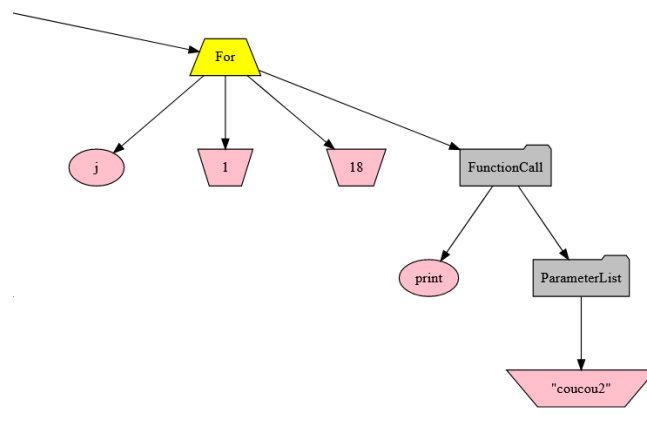


FIGURE 2 – Un morceau d'AST de for avant traitement

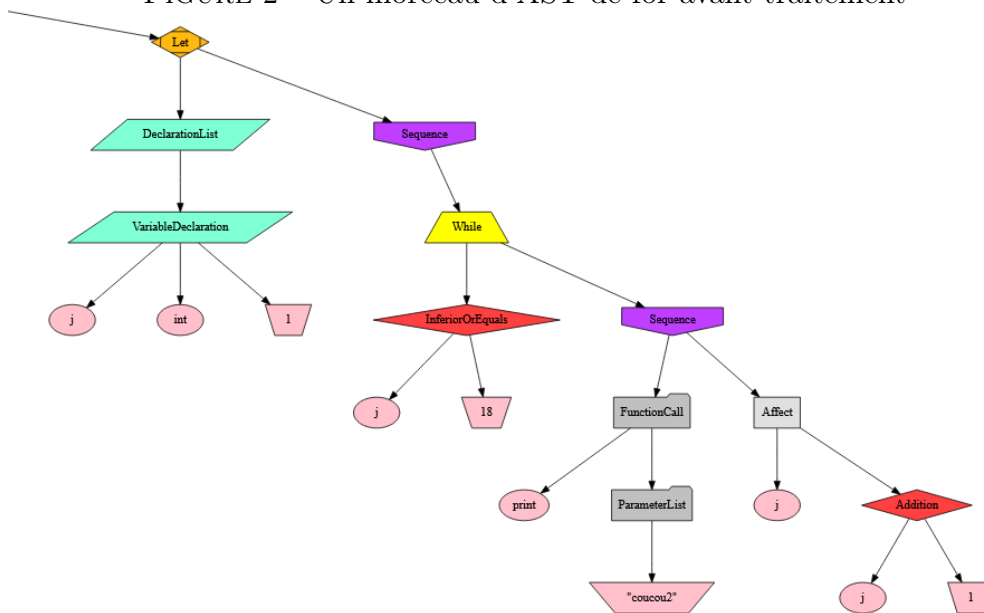


FIGURE 3 – L'AST modifié après traitement

**Gestion des enregistrements** Les valeurs que l'on stocke en assembleur ne sont pas fortement typées. Fort de ce constat, nous avons réalisé que les enregistrements (*record*) sont sémantiquement équivalents à des tableaux sans typage. Nous avons donc pris la décision de transformer le *record* en tableau dans notre AST. L'opération ayant lieu après les contrôles sémantiques, on ne perd pas le respect des règles du langage.

L'opération se découpe en plusieurs étapes :

- Une déclaration de type de record (e.g. `type point = {x:int, y:int, name:string}`) devient une déclaration de type de tableau de *pointer* (e.g. `var point = array of pointer`). On en profite pour associer chaque nom de champs à un indice dans la représentation de ce tableau dans la TDS.
- Pour les déclarations de variable de ce type (e.g. `var p1 := point{x=1, y=2, name="p1"}`), on crée une déclaration de tableau, suivi d'une liste d'affectations.
- Tous les accès de record (e.g. `: p.y`) deviennent des accès de tableau (e.g. `: t[1]`).

**Optimisations statiques** Nous nous sommes servis de ce visiteur pour réaliser en outre un certain nombre d'optimisations statiques :

- **Les expressions arithmétiques** sont simplifiées autant que possible en les remplaçant directement par leur résultat.
- Toute **séquence** n'ayant qu'un seul enfant est éludée. Autrement dit, toute parenthèse inutile est supprimée, cela permet encore une fois de simplifier les expressions arithmétiques.
- **Les expressions booléennes** sont simplifiées autant que possible en respectant les règles du langage sur `and` et `or`. Ainsi avec  $x$  une variable du bon type on a les simplifications d'expressions booléennes suivantes :
  - `0 and x`  $\rightarrow$  `0`
  - `1 and x`  $\rightarrow$  `x`
  - `1 and 1`  $\rightarrow$  `1`
  - `0 and 0`  $\rightarrow$  `0`
  - `0 or x`  $\rightarrow$  `x`
  - `1 or x`  $\rightarrow$  `1`
  - `1 or 1`  $\rightarrow$  `1`
  - `0 or 0`  $\rightarrow$  `0`
- Un `while` dont le condition initiale est évaluée statiquement à *false* est supprimée de l'AST.



## 2.3 Traduction du compilateurs des structures du langage

### 2.3.1 Nouveau scope, appel de fonction et récursivité

Nous gérons l'appel d'une fonction et la création d'un nouveau scope de la même manière :

```

push  CS
push  CD
mov   CD,   SP
CCS           // Calcul et push le nouveau CS
add   CS,   SP,   #16
push  arg1     // Dans le cas d'un Let, on push
push  arg2     // autant de 0 qu'il y a de
push  arg3     // variables dans le scope
...
bl  fname     // APPEL (x30 := pc+4)
push  x30

```

Ainsi, la fin du scope ressemble à :

```

pop   x7       // Résultat au sommet de la pile
pop   x30
push  x7       // Résultat au sommet de la pile
ret           // FIN APPEL (pc := x30)
pop   x7       // Résultat au sommet de la pile
mov   SP,   CD
pop   CD
pop   CS
push  x7       // Résultat au sommet de la pile

```

**CCS : Calcul et push le nouveau CS** Dans le cas d'un Let, le nouveau CS est l'adresse de l'ancien CS, dans le cas d'un appel non récursif il s'agit de l'adresse du scope de définition de la fonction et dans le cas d'un appel récursif il s'agit du même CS.

**Dessin de la pile** La pile ressemble donc à ça :

-2	Ancien CS
-1	Ancien CD
0	Nouveau CS
1	arg1
2	arg2
...	
$n$	@ret
$n + 1$	Exécution puis résultat

TABLE 1 – Pile lors de l'entrée dans un nouveau scope (le trait horizontal indique l'adresse pointée par le nouveau CD)

### 2.3.2 Structures conditionnelles imbriquées

Dans la structure conditionnelle :

- On commence par évaluer la condition et d'empiler le résultat dans la pile
- On dépile le résultat dans un registre temporaire x1
- Suivant la valeur de x1, on branche soit vers le bloc de *\_then\_id* ou celui de *\_else\_id*.
- Après l'exécution des instructions dans l'un des blocs, on termine toujours par un branchement vers *\_end\_id* qui présente la fin de la structure conditionnelle courante

Pour gérer le cas de la présence de plusieurs imbrications, on s'est servi d'une pile temporaire et un *compteur\_id\_if* initialisé à 1 et qui est empilé. Lors de début de la structure conditionnelle, on utilise toujours la valeur de cette variable valeur d'*id*, ce qui qualifie les blocs la même profondeur. Au cas où on arrive à un ifThenElse plus profond, on commence par incrémenter d'un cran *compteur\_id\_if* et empiler cette valeur dans la pile. Enfin, à chaque fois on finit le bloc de *\_end\_id* d'une structure conditionnelle, on dépile dans *compteur\_id\_if*. On note que cette méthode a été utilisée aussi pour gérer les imbrications des boucles.

### 2.3.3 Enregistrements

Les enregistrements sont convertis en tableaux dont la longueur est le nombre d'attributs et à chaque attribut est associé un déplacement qui donne l'indice de l'élément dans le tableau.

### 2.3.4 Tableaux

Un tableau de taille  $N$  est représenté par une structure linéaire de  $N + 1$  cases de 64 bits. La première case contient la longueur du tableau (pour les contrôles sémantiques) et la case  $i + 1$  contient la données d'indice  $i$  dans le tableau.

### 3 Annexe – Un exemple de programme : Fibonacci

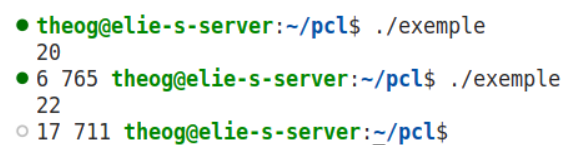
Dans cette partie nous montrerons l'application de notre compilateur avec un programme simple demandant à l'utilisateur saisir un entier N et affichant la terme N de la suite de Fibonacci.

#### 3.1 Code tiger

```
let
  var N := 0
  function fibonacci(n: int): int = (
    if(n=0) then
      (0)
    else(
      if(n=1) then
        (1)
      else
        (fibonacci(n-1) +fibonacci(n-2))
    )
  )
in
  N := input();
  printi(fibonacci(N))
end
```

#### 3.2 Exemple de résultat

Voir la figure 4



```
● theog@elie-s-server:~/pcl$ ./exemple
20
● 6 765 theog@elie-s-server:~/pcl$ ./exemple
22
○ 17 711 theog@elie-s-server:~/pcl$
```

FIGURE 4 – Affichage dans le terminal

#### 3.3 AST de la fonction

Voir la figure 5

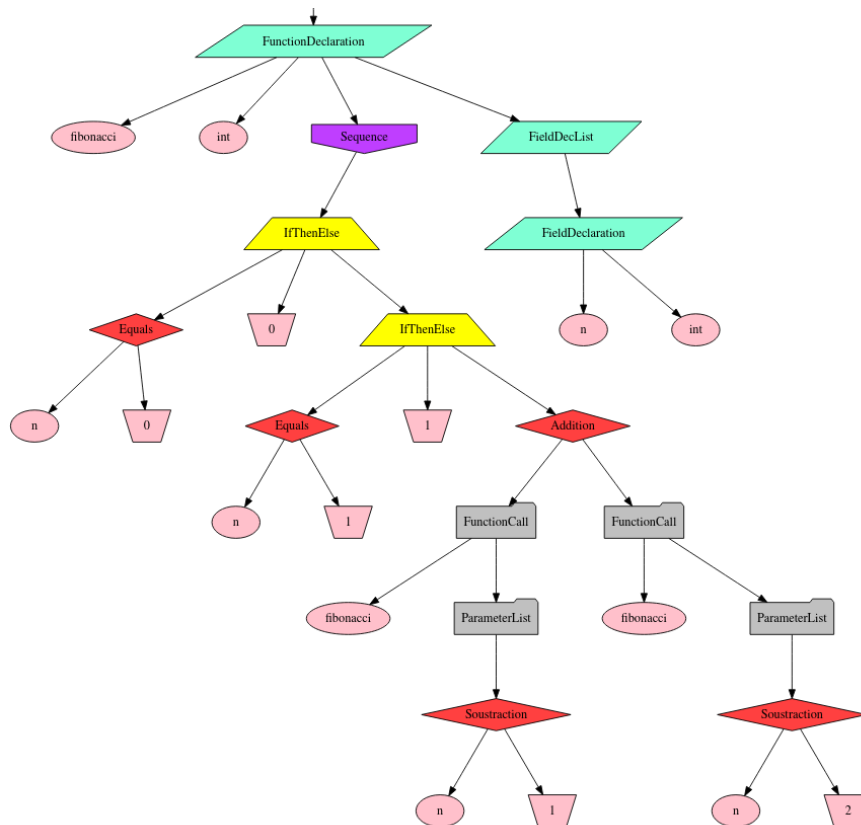


FIGURE 5 – Morceau d'AST correspondant à la fonction fibonacci

### 3.4 Extrait de code tiger

Extrait du code ARM64 correspondant à la fonction fibonacci

```
// BEGIN FUNCTIONS
// BEGIN Function fibonacci
function_7:
push x30/*@retour*/ // @retour
// BEGIN Sequence
// BEGIN IfThenElse2
// BEGIN Operateur binaire
// BEGIN Id n
push x28/*Ch. STAT*/
mov x0, #0 // depth
push x0
```

```
    mov x0, #32 // depl
    push x0
    bl chainage_st // [3] -> [1]
    at // i = *i

// BEGIN IntegerNode
    MOV x9, #0
    push x9

    bl ari_int_EQ // [2] -> [1]
// END Operateur binaire

pop x1
cmp x1,#0
beq _else_2
bne _then_2
_else_2:
// BEGIN IfThenElse3
// BEGIN Operateur binaire
// BEGIN Id n
    push x28/*Ch. STAT*/
    mov x0, #0 // depth
    push x0
    mov x0, #32 // depl
    push x0
    bl chainage_st // [3] -> [1]
    at // i = *i

// BEGIN IntegerNode
    MOV x9, #1
    push x9

    bl ari_int_EQ // [2] -> [1]
// END Operateur binaire

pop x1
cmp x1,#0
beq _else_3
bne _then_3
_else_3:
```

```
// BEGIN Operateur binaire
// BEGIN FunctionCall --- fibonnaci(n-1)
// GESTION DU NOUVEAU SCOPE
push x28/*Ch. STAT*/
push x29/*Ch. DYN*/
mov x29/*Ch. DYN*/, SP/*STACK*/ // Ch. DYN
push x28/*Ch. STAT*/ // Appel récursif
add x28/*Ch. STAT*/, SP, #16 // Ch. STAT
// BEGIN ParameterList
// BEGIN Operateur binaire
// BEGIN Id n
push x28/*Ch. STAT*/
mov x0, #1 // depth
push x0
mov x0, #32 // depl
push x0
bl chainage_st // [3] -> [1]
at // i = *i

// BEGIN IntegerNode
MOV x9, #1
push x9

bl ari_int_sub // [2] -> [1]
// END Operateur binaire
// END ParameterList

bl function_7
// GESTION FIN DU NOUVEAU SCOPE
pop x7 // RES
mov SP/*STACK*/, x29/*Ch. DYN*/ // Ch. DYN
pop x29/*Ch. DYN*/ // Ch. DYN
pop x28/*Ch. STAT*/ // Ch. STAT
push x7 // RES
// END FunctionCall

// BEGIN FunctionCall --- fibonnaci(n-2)
.... IDEM ....
// END FunctionCall
```

```
        bl ari_int_add // [2] -> [1]
    // END Operateur binaire

    b _end_ifthenelse_3
    _then_3:
    // BEGIN IntegerNode
        MOV x9, #1
        push x9

    b _end_ifthenelse_3
    _end_ifthenelse_3:
    // END IfThenElse3

    b _end_ifthenelse_2
    _then_2:
    // BEGIN IntegerNode
        MOV x9, #0
        push x9

    b _end_ifthenelse_2
    _end_ifthenelse_2:
    // END IfThenElse2

    // END Sequence

    pop x7 // RES
    pop x30/*@retour*/ // @retour
    push x7 // RES
    ret
    // END Function fibonacci
```