Rapport PCL2
Potabarnak

Pierre GUYOT, Pierre-Yves JACQUIER,
Arnaud KRAFFT, Titouan LANGLAIS

Sommaire :

I - Introduction

Potabarnak est compilateur de CanAda en Java. Le programme génère de l'ARM64.

II - Table des symboles

II.1 - Principes

La table des symboles (abrégé en TDS) est un composant essentiel dans notre compilateur. Elle permet à la fois de bien réaliser les contrôles sémantiques ainsi que de préparer le terrain pour la traduction. Pour réaliser la TDS, nous récupérons un arbre abstrait structuré en objet Java. Chaque nœud de l'arbre étant un objet attitré à une fonctionnalité. Pour chaque nœud pouvant créer de nouvelles variables (fonction, procédure et boucle for), on crée une sous-TDS comportant les définitions de variables et de fonctions au niveau de son imbrication. De plus, ces sous-TDS sont liées entre elles par rapport à la structure du code. La TDS peut donc être représentée par l'ensemble des sous-TDS, ou bien la sous-TDS racine (la procédure principale).

II.2 - Structures

Quant à elles, les sous-TDS sont une classe Java composée de plusieurs attributs :
- La liste de ses sous-TDS filles
- Sa sous-TDS parent (égale à `null` dans le cas de la sous-TDS racine)
- Le nœud associé à cette dernière (la procédure principale, la définition d'une fonction ou procédure, une boucle for)
- Son numéro d'imbrication (0 pour la sous-TDS racine)
- Un numéro unique de région (0 pour la sous-TDS racine)
- Son nom et des informations supplémentaires sur son type (fonction / procédure)
- La liste de ses variables et de ses paramètres (ne comporte pas les variables des sous-TDS fille ou mère)
- La liste des fonctions et procédures que l'on peut utiliser dans cette sous-TDS (la fonction ou procédure de la sous-TDS, ainsi que les fonctions et procédures définis dans la sous-TDS)

II.3 - Algorithme de création

En premier lieu, les sous-TDS sont créées récursivement en parcourant l'arbre. À chaque nœud spéciaux (procédure principale, fonction et procédure et boucle for), une nouvelle sous-TDS est créée et cherche toute les variables et fonctions / procédures déclarées en son sein, mais pas dans ses sous-TDS filles ou parente. L'ajout de variables dans la sous-TDS distingue automatiquement pour la génération de code, si cette dernière est un paramètre, ou une déclaration.

Après la génération des sous-TDS, on s'occupe de donner à chaque nœud de l'arbre, un lien vers la sous-TDS dont il dépend, afin de faciliter les contrôles sémantiques et la génération de code nœud par nœud. Enfin, une dernière étape se charge de lier les variables et les fonctions / procédures utilisées dans des appels ou autres utilisations, à une variable ou fonction déjà déclarée dans les sous-TDS. Pour choisir la variable à associer, on regarde si une variable possède le même nom dans la sous-TDS associée au nœud, sinon on remonte à la sous-TDS parente. Si aucune variable n'est trouvée, cela remonte une erreur. Pour les fonctions et procédures, c'est la même chose, si on ne trouve pas de définitions, on remonte à la TDS parente.

II.4 - Exemple

Pour l'exemple de la structure de la TDS, nous avons choisi d'utiliser ce programme qui montre à la fois la structure des 4 types de TDS (procédure principale, fonction / procédure et boucle for), ainsi que leur imbrication. Pour cela, nous avons écrit un programme composé d'une fonction avec un paramètre et une variable, une procédure composée d'une définition de fonction similaire à la précédente en plus d'un paramètre et d'une variable. Et enfin, une boucle for utilisant une variable `i`.

Ainsi, le programme suivant donnera la TDS ci-dessous, représentée sous forme d'arbre de tableaux, avec les différentes informations en son sein.

```
with Ada.Text_IO ; use Ada.Text_IO ;

procedure TDS_Showcase is

    function fonction_A(parametre_A : Integer) return Integer is
        variable_A : Integer := 0;
    begin
        return variable_A + parametre_A;
    end fonction_A;

    procedure procedure_B(parametre_B : Integer) is
        variable_B : Integer := 0;

        function fonction_C(parametre_C : Integer) return Integer is
            variable_C : Integer := 0;
        begin
            return variable_C - parametre_C;
        end fonction_C;

    begin
        variable_B := fonction_A(parametre_B);
        variable_B := variable_B + fonction_C(parametre_B);
        put(variable_B);
    end procedure_B;

begin

    procedure_B(5);
    for i in 1..10 loop
        fonction_A(i);
        put(fonction_A(i));
    end loop;

end TDS_Showcase;
```

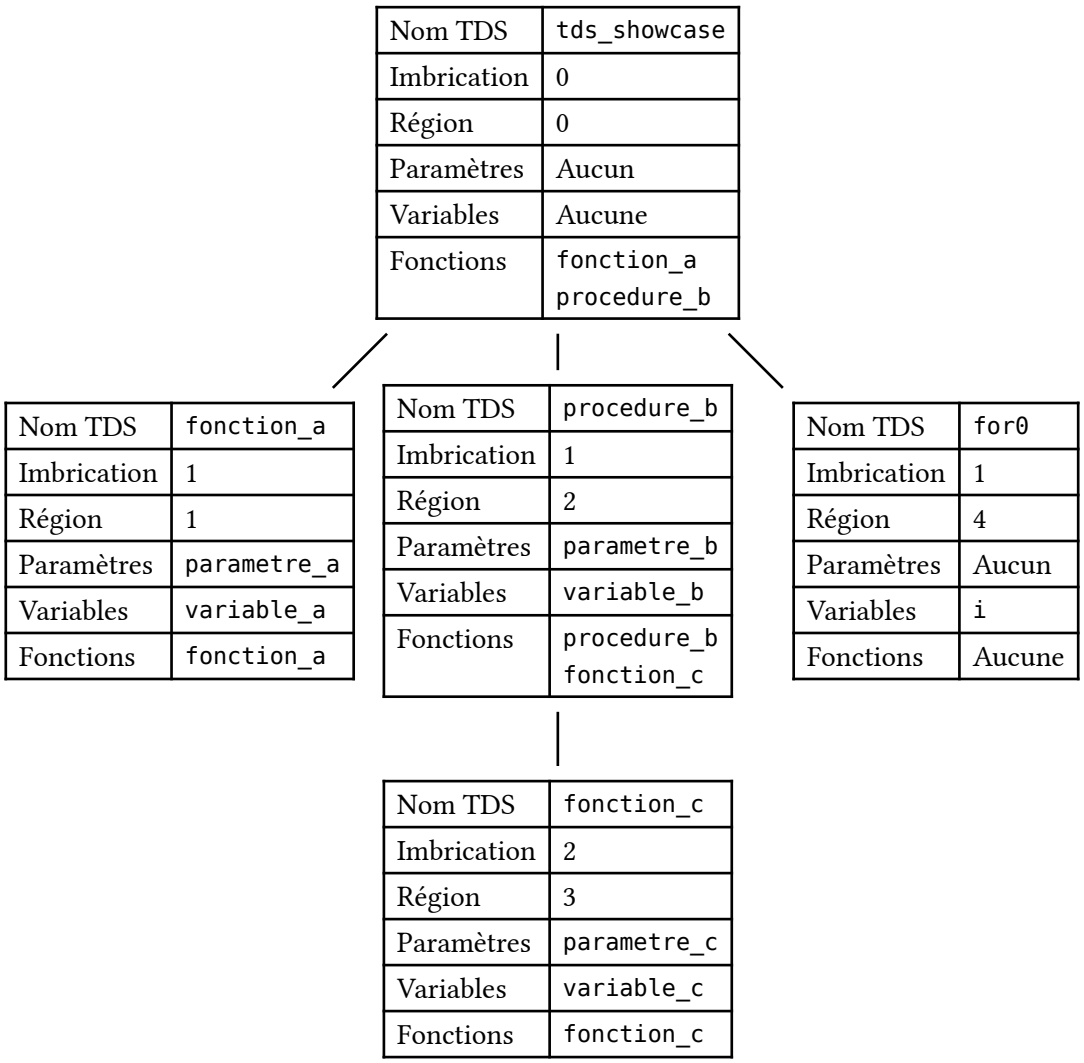


Fig. 1. – Table des symboles du programme `TDS_Showcase`

III - Contrôles sémantiques

III.1 - Contrôles de cohérence

Les contrôles sémantiques de « cohérence » vérifient que les instructions du programme ne posent pas de problème, concernant principalement les types et les appels de fonction.

Par exemple, il est impossible d'effectuer une opération entre un entier et un caractère ou d'affecter un booléen à un entier. Pour les appels de fonction, on vérifie que le nombre de paramètres et leurs types sont corrects.

La vérification se fait principalement à l'aide d'un appel récursif de la méthode `valide()` que chaque nœud de l'arbre syntaxique possède. Un nœud vérifiera donc sa validité en regardant la validité de ses enfants et ainsi de suite.

Une partie des contrôles sémantiques sont aussi « intégrés » à la structure de l'AST et sont donc réalisés lors de sa construction, par exemple, une affectation demande une variable comme membre gauche dans tous les cas, il n'est donc pas nécessaire d'ajouter un contrôle supplémentaire pour s'assurer qu'on n'affecte pas une valeur à une constante.

III.2 - Contrôles utilisant la table des symboles

Les contrôles sémantiques utilisant la table des symboles restent assez simple. Les premiers sont des vérifications sur l'unicité des noms de variables et de fonctions dans chacune des sous-TDS. Puis ensuite, on vérifie si tous les appels de fonctions et utilisations des variables utilisent une fonction ou une variable bien déclarée dans la sous-TDS parent (ou parent à cette dernière).
Si tous ces contrôles sémantiques ne comportent aucune erreur, alors on peut passer à la partie génération de code.

IV - Génération de code

IV.1 - Principe

La génération de code est le processus permettant de passer de l'AST au programme dans le langage cible. Pour cela, nous avons une hypothèse simplificatrice : L'AST que nous traitons est « juste ». Cela veut dire qu'il n'y a ni erreur syntaxique, ni erreur lexicale dans le programme source.

Nous avons fait le choix de la simplicité de programmation avant la rapidité ou l'efficacité du programme. De ce fait, nous avons choisi d'utiliser la pile le plus possible. En effet, l'utilisation des registres demande une gestion du nombre de registres restants, alors que la pile ne demande presque aucune gestion de la place restante. Toujours dans un souci de simplicité, le décalage dans la pile est de taille fixe et de la taille du plus grand objet possible, dans notre cas 64 bits.

Nous utilisons l'ARMv8 AARCH64.

Ce qui nous donne comme utilisation de registres :
- x0 : Registre de calcul et de résultat
- x1 : Registre de calcul
- x2 : Registre utilisé lors de l'affectation ($a :=$)
- x8 : Registre utilisé pour les appels système
- x26 : Registre de valeur de retour des appels de fonctions
- x27 : Registre de sauvegarde du chaînage statique
- x28 : Registre utilisé pour sauvegarder le chaînage statique lors de la récupération d'une variable globale
- x29 : Frame pointer
- x30 : Adresse de retour
- sp : Stack pointer

IV.2 - Les instructions

Pour les instructions basiques du type $a := B;$, nous traitons d'abord le membre de droite, puis le membre de gauche, respectivement B et a dans l'exemple.
B peut prendre plusieurs formes différentes :
- X `op` Y
- `op` X
- a

Avec B, X, Y des compositions d'évaluables, `op` un opérateur et a un évaluable. (i.e. un élément pouvant avoir une valeur (constante, variable, appel de fonction, opération))

Nous avons décidé de mettre tout ce dont nous avons besoin en pile. De cette façon, à chaque fois que nous avons besoin de quelque chose, par exemple le résultat d'un calcul, il est en tête de pile.

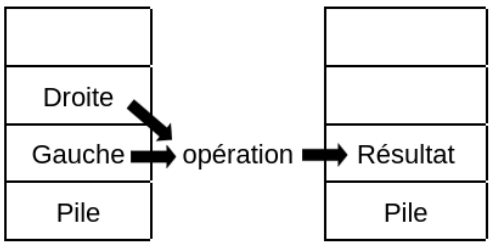


Fig. 2. – Utilisation de la pile avec un opérateur binaire

Une fois le résultat en tête de pile, nous pouvons le mettre dans la variable en utilisant les Tables des Symboles TDS (Chapitre II), ici la variable est a.

IV.2.a - Exemples de traduction

```
z:= 5;
y:= 3;
j := y + z;
```

Est traduit en :

```
MOVZ x0, #5
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-80] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur

MOVZ x0, #3
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-64] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur

// Opération
LDR x0, [x29, #-64] // On récupère la valeur de la variable y
SUB sp, sp, #16 // y Mise en pile var
STR x0, [sp] // y Mise en pile var
LDR x0, [x29, #-80] // On récupère la valeur de la variable z
SUB sp, sp, #16 // z Mise en pile var
STR x0, [sp] // z Mise en pile var
LDR x1, [sp] // On met l'opérande droite dans x1
ADD sp, sp, #16 // On décrémente le pointeur de pile
LDR x0, [sp] // On met l'opérande gauche dans x0
ADD sp, sp, #16 // On décrémente le pointeur de pile
ADD x0, x0, x1 // Opération +
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met le résultat en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-48] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur
```

Dans le cas présent, aucun test n'est effectué lors du calcul mais il est possible d'en avoir. En effet, nous devons vérifier le cas des divisions par 0.

```
a := z / y;
```

Est traduit en :

```
LDR x0, [x29, #-64]
SUB sp, sp, #16
STR x0, [sp]
LDR x0, [x29, #-80]
SUB sp, sp, #16
STR x0, [sp]
LDR x1, [sp]
ADD sp, sp, #16
LDR x0, [sp]
ADD sp, sp, #16
CMP x1, #0 // Vérification du membre droite différent de 0
BEQ erreur_division // Saut vers la fonction erreur pour arréter le programme
SDIV x0, x0, x1 // La division
SUB sp, sp, #16
STR x0, [sp]
LDR x2, [sp]
STR x2, [x29, #-48]
ADD sp, sp, #16
```

IV.3 - Les fonctions & les procédures

La différence majeure entre les fonctions et les procédures est la valeur retournée. Ainsi, dans la génération de code, la différence majeure est le système de sortie du bloc, pour les fonctions; la sortie est générée par le `return` alors que pour les procédures, la sortie est à la fin du bloc ‹ procédure ›. Par conséquent, nous ne ferons pas de distinction entre procédure et fonction dans le reste de la partie.

IV.3.a - La définition des fonctions

Les fonctions sont des blocs qui agglomèrent et génèrent leur propre code. En effet, une instruction (Chapitre IV.2) peut-être générée dans la procédure ‹ main › ou dans les fonctions. Pour éviter tout problème, les fonctions sont nommées F + leur numéro de région (resp. P pour les procédures), cette fonctionnalité nous permet de lire le programme assembleur plus facilement et évite tout problème de fonctions avec le même nom. Par exemple, on peut définir une fonction ‹ erreur_division › alors que cette fonction est utilisée en assembleur pour gérer les cas de division par 0.

Toutes les fonctions, sont générées à la suite à la fin du programme.

IV.3.b - Les appels de fonctions

Lors d'un appel de fonction en Ada, nous avons besoin de plusieurs choses :

- Les paramètres : mis en place par l'appelant, en effet les paramètres peuvent dépendre de variables à l'intérieur de la fonction appelante.
- Le chaînage statique : mis en place par l'appelant. Il sert à la récupération des variables globales. Pour pouvoir sélectionner l'adresse que l'on met dans le chaînage statique, il nous faut savoir si la fonction appelée est la même que la fonction appelante, or dans notre modélisation de l'AST, il est plus simple de connaître les fils que les pères. Ainsi, le chaînage statique est mis en place par l'appelant.
- Le chaînage dynamique : mis en place par l'appelé. Il sert au stockage du ‹ frame pointer ›, qui nous sert à restaurer la pile lors de la sortie de la fonction.
- L'adresse de retour : mis en place par l'appelé. Elle nous sert à stocker l'adresse de l'instruction qui sera exécutée après la sortie de la fonction.
- Les variables locales : mis en place par l'appelé.

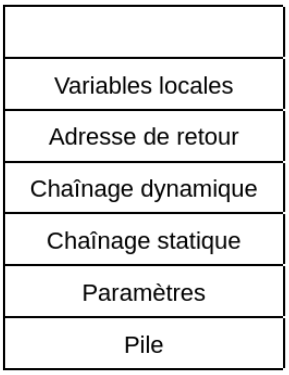


Fig. 3. – Utilisation de la pile durant un appel de fonction

Le bloc ‹ return › met la valeur de retour dans un registre spécialement réservé, et le programme appelant dépile ce qui a été empilé et empile la valeur de retour.

IV.3.c - Récupération des variables globales

Pour récupérer les variables globales, on utilise les chaînages statiques ainsi que les TDS. On fait un appel de fonction à la fonction ‹ get_global_var › ou ‹ set_global_var › avec le déplacement depuis le frame pointer dans x0 et le nombre d'imbrications à remonter dans x1.

Le programme ci-dessous est utilisé à chaque fois que l'on veut accéder à une variable globale.

```
MOVZ x0, #64 // Deplacement en pile VAR GLOBALE
MOVZ x1, #1 // b Nb saut VAR GLOBALE
MOV x28,x29 // Copie du frame pointer dans x28 (temporaire)
BL get_global_var // b Mise en pile var
```

Le programme ci-dessous est généré au début de la compilation et peut-être utilisé par toutes les fonctions.

```
get_global_var : LDR x28, [x28] // On saute de chainage statique, x0 depl, x1 nb_saut
SUBS x1, x1, #1 // On décrémente le nombre de saut
BNE get_global_var // On boucle tant que x1 != 0
SUB x28, x28, x0 // On déplace le pointeur de la variable
LDR x0, [x28] // On charge la valeur de la variable
SUB sp,sp, #16 // On fait de la place dans la pile pour le retour
STR x0, [sp] // On met la valeur de la variable en pile
RET

set_global_var : LDR x28, [x28] // On saute de chainage statique, x0 depl, x1 nb_saut
SUBS x1, x1, #1 // On décrémente le nombre de saut
BNE set_global_var // On boucle tant que x1 != 0
SUB x28, x28, x0 // On déplace le pointeur de la variable
STR x2, [x28] // On charge la valeur de la variable
RET
```

IV.3.d - Exemples de traduction

IV.3.d.a - Appel de fonction

```
    a := Add100(a,515);
```

Est traduit en :

```
// Appel de fonction add100
// Paramètre 0
```

```
MOVZ x0, #515
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
// Paramètre 1
LDR x0, [x29, #-48] // On récupère la valeur de la variable a
SUB sp, sp, #16 // a Mise en pile var
STR x0, [sp] // a Mise en pile var
// Gestion du chainage statique
SUB sp, sp, #16 // Incrémentation du pointeur de pile
STR x29, [sp] // Sauvegarde du chainage statique
MOV x27, x29 // Mise à jour du chainage statique
BL F1 // Appel de la fonction
// Gestion du chainage statique
ADD sp, sp, #16 // Le chainage statique ça dégage
// Récupération du résultat
ADD sp, sp, #32 // Décrémentation du pointeur de pile de la taille des paramètres
SUB sp, sp, #16 // Réserve de l'espace pour le résultat
STR x26, [sp] // Sauvegarde du résultat
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-48] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur
```

IV.3.d.b - Définition de fonction

```
function Add100(x: Integer; x2: Integer) return Integer is

y: Integer := x;

begin
    x := x + 100;
    return x;
end Add100;
```

Est traduit en :

```
F1 : // Début de la fonction
STP x29, lr, [sp, #-16] // Sauvegarde du pointeur de pile et du lien de retour
MOV x29, sp // Mise à jour du pointeur de pile
SUB sp, sp, #32 // Déplacement du stack pointer pour fp et lr
// Définitions de la fonction add100
// Declaration de la variable y
LDR x0, [x29, #16] // On récupère la valeur de la variable x
SUB sp, sp, #16 // x Mise en pile var
STR x0, [sp] // x Mise en pile var

// Instructions de la fonction add100

// Opération
LDR x0, [x29, #16] // On récupère la valeur de la variable x
SUB sp, sp, #16 // x Mise en pile var
STR x0, [sp] // x Mise en pile var
MOVZ x0, #100
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x1, [sp] // On met l'opérande droite dans x1
ADD sp, sp, #16 // On décrémente le pointeur de pile
LDR x0, [sp] // On met l'opérande gauche dans x0
ADD sp, sp, #16 // On décrémente le pointeur de pile
```

```
ADD x0, x0, x1 // Opération +
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met le résultat en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #16] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur

// Return
LDR x0, [x29, #16] // On récupère la valeur de la variable x
SUB sp, sp, #16 // x Mise en pile var
STR x0, [sp] // x Mise en pile var
LDR x26, [sp] // Valeur de retour dans le registre x26
MOV sp, x29 // Restauration du pointeur de pile
LDP x29, lr, [sp, #-16] // Restauration du pointeur de pile et du lien de retour
RET // Retour de la fonction
```

IV.4 - Les instructions if, while et for

Ces instructions utilisent des étiquettes uniques pour fonctionner à l'aide de branchements. Par conséquent, il faut générer un ID unique pour chacune de ces instructions (pour éviter qu'un if branche sur un autre etc). Pour cela, on utilise la fonction `hashCode()` de Java, donnant un entier unique pour chaque objet.

IV.4.a - Instruction If

La génération de code des instructions if utilise des blocs précédents pour fonctionner. D'abord, on génère l'opération de la condition (constante, opération, appel de fonction...), ce résultat est toujours mis en haut de la pile, par conséquent, on peut le charger et le comparer à 0.

Si le résultat n'est pas égal à 0, alors on branche vers la zone dédiée au then. Après le branchement se trouve les instructions du else s'il y en a un ou rien sinon, dans tous les cas, il y a en suite un branchement vers la zone `end`, qui permet d'éviter d'exécuter le code du then si la condition n'est pas remplie.

IV.4.b - Instruction While

Les instructions while possèdent une logique similaire au if. Elles sont divisées en trois étiquettes : `while`, `whilecontinue` et `whileend`. `while` est dédiée à l'évaluation de la condition et de sa comparaison avec 0, comme pour le if, si ce n'est pas égal on branche sur `whilecontinue`, sinon sur `whileend`. À la fin de `whilecontinue`, on réalise un branchement vers `while` pour réévaluer la condition de boucle.

IV.4.c - Instruction For

Les instructions for sont entièrement construites à partir d'autres blocs de construction présents dans le compilateur. En fait, lors de la production, le compilateur va traduire une boucle for en sa boucle while équivalente, puis produire cette boucle while.

IV.4.d - Exemples de traduction

IV.4.d.a - Instruction If

```
if a = 5 then
    b:= 6;
else
    b:= 7;
end if;
```

Est traduit en :

```
// Opération
LDR x0, [x29, #-48] // On récupère la valeur de la variable a
SUB sp, sp, #16 // a Mise en pile var
STR x0, [sp] // a Mise en pile var
MOVZ x0, #5
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x1, [sp] // On met l'opérande droite dans x1
ADD sp, sp, #16 // On décrémente le pointeur de pile
LDR x0, [sp] // On met l'opérande gauche dans x0
ADD sp, sp, #16 // On décrémente le pointeur de pile
CMP x0, x1 // Opération =
CSET x0, EQ // Opération =
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met le résultat en pile
LDR x0, [sp] // Chargement de la condition
ADD sp, sp, #16 // Décrémentation du pointeur de pile
CMP x0, #0 // Comparaison de la condition
BNE then1104106489 // Branchement si la condition est vraie
MOVZ x0, #7    //// Le "Else" commence ici !
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-64] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur
B end1104106489 // Branchement à la fin du if
then1104106489 :    //// Le "Then" commence ici !
MOVZ x0, #6
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-64] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur
end1104106489 :
```

IV.4.d.b - Instruction While

```
    while a /= 0 loop
        a:= a - 1;
    end loop;
```

Est traduit en :

```
// while 1104106489
while1104106489 :

// Opération
LDR x0, [x29, #-48] // On récupère la valeur de la variable a
SUB sp, sp, #16 // a Mise en pile var
STR x0, [sp] // a Mise en pile var
MOVZ x0, #0
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x1, [sp] // On met l'opérande droite dans x1
ADD sp, sp, #16 // On décrémente le pointeur de pile
LDR x0, [sp] // On met l'opérande gauche dans x0
ADD sp, sp, #16 // On décrémente le pointeur de pile
CMP x0, x1 // Opération /=
```

```
CSET x0, NE // Opération /=
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met le résultat en pile
LDR x0, [sp] // Chargement de la condition
ADD sp, sp, #16 // Dépilement de la condition
CMP x0, #0

BNE whilecontinue1104106489
B whileend1104106489
whilecontinue1104106489 :

// Opération
LDR x0, [x29, #-48] // On récupère la valeur de la variable a
SUB sp, sp, #16 // a Mise en pile var
STR x0, [sp] // a Mise en pile var
MOVZ x0, #1
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met la constante en pile
LDR x1, [sp] // On met l'opérande droite dans x1
ADD sp, sp, #16 // On décrémente le pointeur de pile
LDR x0, [sp] // On met l'opérande gauche dans x0
ADD sp, sp, #16 // On décrémente le pointeur de pile
SUB x0, x0, x1 // Opération -
SUB sp, sp, #16 // On décrémente le pointeur de pile
STR x0, [sp] // On met le résultat en pile
LDR x2, [sp] // On met la valeur de la variable droite dans x2
STR x2, [x29, #-48] // On met la valeur de la variable droite dans la variable gauche
ADD sp, sp, #16 // On dépile la valeur
B while1104106489
whileend1104106489 :
```

IV.4.d.c - Instruction For

Les boucles For étant transformées en boucles While, un exemple de traduction serait redondant.

IV.5 - Parties non-traités

Par manque de temps, nous avons préféré ne pas traiter certaines parties de la grammaire :

- In & In-out : Notre implémentation actuelle des fonctions ne nous permet pas de traiter cette partie rapidement, nous l'avons donc mise de côté.
- Record et access : Nous avions prévu de les faire à la fin, nous avons préféré ne pas les traiter par manque de temps.

V - Programme de test final

Le programme de test final est un programme permettant de calculer la différence entre la suite de Fibonacci et la suite de Tribonacci pour chaque valeur de 2 à n, n étant une valeur fixée. Il permet également de calculer l'accélération de cette différence.

La suite de Tribonacci repose sur le même principe que la suite de Fibonacci, si ce n'est que le résultat de la suite en n+3 est égal à l'addition des résultats en n+2, n+1 et n, contrairement à Fibonacci où le résultat est égal à l'addition des résultats en n+1 et n.

Le programme comporte donc des déclarations de variable, de fonctions et de procédures, ainsi que des appels de fonctions à l'intérieur d'appels de procédures. Il y a également des boucles « for » et « if ».

Il est à noter que la version de l'algorithme de la suite de Fibonacci utilisé est une version permettant de n'avoir qu'un seul appel récursif par appel de la fonction pour accélérer l'exécution.

L'exécution du programme imprime donc pour chaque entier i de 2 à n :
- La valeur en i de la suite de Fibonacci.
- La valeur en i de la suite de Tribonacci.
- La différence des deux valeurs précédentes.
- L'accélération de cette différence.

```
with Ada.Text_IO;use Ada.Text_IO;

procedure diff_tribo_fibo is

    n: Integer;

    procedure acc_fib_trib(n: Integer) is

        trib: Integer;
        fib: Integer;
        f0: Integer;
        f1: Integer;
        f2: Integer;

        function fibonacci(n: Integer; x1: Integer; x2: Integer) return Integer is
            begin
                if n = 0 then
                    return x1;
                else
                    return fibonacci(n - 1, x2, x1 + x2);
                end if;
                return 0;
            end fibonacci;

        function tribonacci(n: Integer; x1: Integer; x2: Integer; x3: Integer) return
Integer is
            begin
                if n = 0 then
                    return x1;
                else
                    if n = 1 then
                        return x2;
                    else
                        if n = 2 then
                            return x3;
                        else
                            return tribonacci(n-1, x2, x3, x1+x2+x3);
                        end if;
                    end if;
                end if;
                return 0;
            end tribonacci;

        function get_acceleration(f0: Integer; f1: Integer; f2: Integer; n1: Integer;
n2: Integer) return Integer is
            begin
                return (f2-2*f1+f0)/((n2-n1)*(n-n1));
```

```
            end get_acceleration;

    begin
        fib := fibonacci(n, 0, 1);
        Put('f');
        Put(fib);
        trib := tribonacci(n, 0, 1, 1);
        Put('t');
        Put(trib);
        Put('d');
        Put(trib-fib);
        if n > 1 then
            f0 := tribonacci(n - 2, 0, 1, 1) - fibonacci(n - 2, 0, 1);
            f1 := tribonacci(n - 1, 0, 1, 1) - fibonacci(n - 1, 0, 1);
            f2 := trib - fib ;
            Put('a');
            Put(get_acceleration(f0, f1, f2, n - 1, n));
        end if;
    end acc_fib_trib;

begin
    n := 30;
    for i in 2..n loop
        Put('n');
        Put(n);
        acc_fib_trib(i);
    end loop;
end diff_tribo_fibo;
```

VI - Gestion de projet

VI.1 - Répartition du travail

Tout au long de cette seconde partie du projet PCL, la répartition du travail a été la suivante:

- Pierre-Yves JACQUIER s'est occupé des contrôles sémantiques de cohérence, de la correction de bugs et d'une partie de la génération de code.
- Pierre GUYOT s'est occupé de l'implémentation de la table des symboles.
- Titouan LANGLAIS s'est occupé de la génération de code assembleur
- Arnaud KRAFFT s'est occupé de rédiger des fichiers de tests (y compris le test final) et d'exemples, de faire du debugging et d'occasionnellement aider dans les trois autres tâches.

VI.2 - Rapport Personnel

VI.2.a - Pierre GUYOT

Même si cette seconde partie était plus complexe dans ses mécaniques et donc plus intéressantes en générale, j'ai eu plus du mal sur la fin du projet. En effet, la gestion des erreurs sémantiques et la création de la table des symboles était un défi captivant en parcourant l'AST, et en essayant de gérer tous les cas possible. Cependant, c'est cette dernière partie qui a été plus difficile pour moi : essayer de peaufiner le programme afin qu'il ne plante pas et qu'il renvoie toutes les erreurs que l'utilisateur pourrait commettre, et prévoir un substitut pour que le compilateur continue sa recherche d'erreurs.

VI.2.b - Pierre-Yves JACQUIER

De mon côté, une petite perte de vitesse comparé à la PCL1 où je m'amusais à implémenter des fonctionnalités pour passer le temps. Un *certain concours de circonstances et de raisons personnelles* a fait que je n'ai pas pu m'investir autant que je voulais dans le projet, ce qui a, je pense, eu des conséquences sur la fluidité et la qualité du projet. Ma plus grande tâche était surtout de peaufiner les détails et de réparer les défauts de notre partie PCL1. Cela, avec la génération de code et les contrôles sémantiques m'a permis de toucher un peu à tout et de collaborer avec tout le monde sur leurs parties respectives ~~bien que je sois totalement allergique au code de Pierre~~.

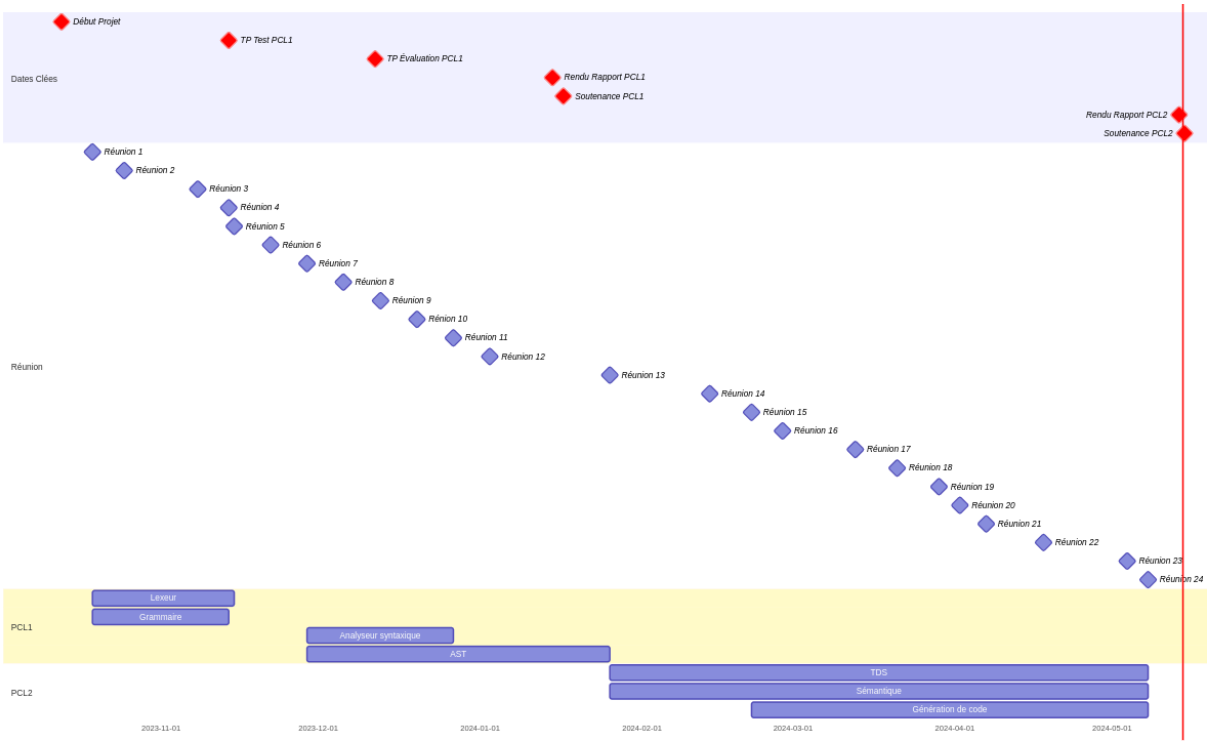
VI.2.c - Arnaud KRAFFT

Je me suis senti moins investi dans cette seconde partie du projet que dans d'autres projets que nous avons eu. J'ai ici plutôt eu un rôle de support : aider les autres dans leurs parties quand ils en avaient besoin, tester leur travail pour trouver des bugs à corriger et des edge cases, faire des programmes de test, etc. Ceci m'a laissé un peu insatisfait. Même si au bout du compte le projet a tout de même avancé à bonne une vitesse, je me suis senti moins utile au groupe.

VI.2.d - Titouan LANGLAIS

Bien que j'ai globalement apprécié mon expérience sur ce projet, plusieurs aspects auraient pu être améliorés pour nous permettre de finir tout ce qui était prévu. Notamment, l'absence d'une gestion de projet structurée a nettement limité notre progression et notre efficacité. J'aurais aimé que nous puissions pousser nos limites plus loin, en explorant plus de possibilités, en optimisant nos résultats, et en visant les records, par exemple. Un autre point frustrant a été notre incapacité à apprendre de nos erreurs précédentes, surtout en ce qui concerne les programmes de test, ce qui a entravé notre progression. Cependant, malgré ces défis, travailler sur des aspects techniques comme l'assembleur reste toujours un plaisir et une source d'enrichissement.

VI.3 - Gantt



VII - Annexes

VII.1 - CR 1

Compte-rendu - 19 octobre 2023 Participants: Tout le monde

Date: 2023-10-19

Lieu: Anim'Est Local

Heure: 12h45 - 13h30

Activités :
PYJ: Réalisation d'un lecteur de caractères en Java PG: Début de la création d'un dictionnaire T & A: Création d'un programme « Hello World » en Ada et d'un programme Fibonacci pour tester la récursivité

Discussion :
Nécessité de lire le fichier, de le transformer à l'aide du dictionnaire et d'en extraire le contenu en supprimant les espaces, les tabulations et les commentaires tout en remplaçant les mots par des nombres. Retravailler la grammaire pour pouvoir la transformer ultérieurement, car elle contient encore des expressions régulières.

Tâches à réaliser:

PG: Création d'un diagramme de Gantt et d'une structure WBS PYJ: Élaboration d'un dictionnaire lexical AK et TA: Refonte de la grammaire

VII.2 - CR 2

Compte-rendu - 25 octobre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-10-25

Lieu: Anim'Est Local

Heure: 11h50

Activités :
Explication du diagramme de Gantt et de la structure WBS Vérification de la grammaire : réduction, dé-récursivisation et factorisation. Nécessite une vérification plus approfondie. Décision de conserver les espaces pour l'analyseur syntaxique (déjà codé en int dans l'analyseur) Pour les unités lexicales simples : copier le tableau. Pour les mots-clés : catégoriser et coder avec le codage de Huffman (à déterminer)

Tâches à faire :
PG: Transformer l'analyseur lexical PYJ: Réaliser la visualisation et le codage de l'arbre en Java AK et TL: Poursuivre le travail sur la grammaire, en particulier les règles « Suivant » et « Premier »

VII.3 - CR 3

Compte-rendu - 8 novembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-11-08

Lieu: Anim'Est Local

Heure: 16h00

Activités :
Grammaire: En cours de développement, avec des récursivités à gauche à corriger. PYJ: Création d'une arborescence de fichiers, avec la mise en place de certains nœuds et la réalisation d'exemples. Le lecteur est fonctionnel mais son apparence reste à améliorer.

Tâches à faire :
PG: Réviser l'analyseur et son lecteur PYJ: Poursuivre la visualisation de l'arborescence AK et TL: Continuer le travail sur la grammaire

VII.4 - CR 4

Compte-rendu - 14 novembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-11-14

Lieu: Tek'TN

Heure: 13h00

Activités : Préparation de la présentation du lendemain:
- La grammaire est de type LL1 mais présente des incohérences :
- Les expressions sont traitées comme des instructions (ex: 2 + 3 = 5)
- Les appels de fonctions et les arrêts de programme sont considérés comme équivalents dans la syntaxe
- Les valeurs L peuvent poser des problèmes supplémentaires (sémantiques)
- Nécessité de relire la grammaire
- PYJ: Ajout de fonctionnalités aux arbres permettant d'effectuer toutes les opérations souhaitées
- Explication du lexeur

Tâches à faire:
Attente de la réunion du lendemain pour la suite des travaux

VII.5 - CR 5

Compte-rendu - 15 novembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-11-15

Lieu: Tek'TN

Heure: 13h11

Activités:
Pierre: Présentation du logger dans l'analyseur lexical

Discussion:
Nécessité de créer un diagramme de classes (MOCI) pour l'analyseur syntaxique Manque de connaissances suffisantes pour réaliser le diagramme de classes Décision de ne pas modifier l'analyseur lexical sauf en cas de crash

Tâches à faire:
Lire le « Dragon Book » Commencer la création du diagramme de classes

VII.6 - CR 6

Compte-rendu - 22 novembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-11-22

Lieu: Anim'Est

Heure: 10h38

Activités:
Lecture du « Dragon Book » : Aucune information utile en plus pour le développement du projet.

Tâches à faire:
Anim'Est: Tâches non précisées, oubliées par le rédacteur du compte-rendu.

VII.7 - CR 7

Compte-rendu - 29 novembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-11-29

Lieu: Tek'TN

Heure: 13h11

Activités:
Arnaud: Création d'une fonction de test de pile pour l'utilisation des piles en Java PYJ: Développement d'un nouveau type de nœud composé uniquement de texte et générant du code PlantUML Titouan: Élaboration d'une table d'analyse, avec des règles commençant par le numéro 1. Nécessité de lister toutes les règles et les non-terminaux par la suite. Modification du dictionnaire lexical

Tâches à faire:
Arnaud: Intégration des terminaux et des non-terminaux Pierre: Intégration des règles Titouan: Finalisation du tableau de la grammaire PYJ: Finalisation des travaux sur l'arborescence

VII.8 - CR 8

Compte-rendu - 6 décembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-12-06

Lieu: 1.15

Heure: 14h06

Activités :
Arnaud: Intégration de 39 non-terminaux et de 61 terminaux PYJ: Finalisation des travaux sur l'arborescence Titouan: Remplissage manuel de la table

Tâches à faire :
Pierre: Lecture et analyse de l'ensemble des éléments (non précisé)

VII.9 - CR 9

Compte-rendu - 13 décembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-12-13

Lieu: Tek'TN

Heure: 14h43

Points discutés:
Tâches restantes pour l'analyseur syntaxique: Définir l'algorithme d'analyse syntaxique Mettre en place les structures de données nécessaires Adapter le lecteur de l'analyse lexicale pour l'analyse sémantique Obtenir un arbre explorable Relier l'analyseur syntaxique à l'arbre abstrait syntaxique (AST) Gérer les erreurs syntaxiques (au niveau du bloc ou de la ligne) Déroulement ordonné des tâches: Mise en place des structures de données (Pierre) Création d'une matrice de passage entre l'analyseur lexical et l'analyseur syntaxique (Arnaud) Développement de l'algorithme d'analyse syntaxique (Pierre) Connexion de l'analyseur syntaxique à l'AST (PYJ) Réalisation de programmes en Ada (Titouan)

Tâches à faire :
(Aucune tâche n'a été définie dans le compte-rendu.)

VII.10 - CR 10

Compte-rendu - 20 décembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-12-20

Lieu: Tek'TN

Heure: 10h32

Activités:
Structures de données: Les structures de données sont créées, mais les entrées ne sont pas encore implémentées. Algorithme d'analyse syntaxique: L'algorithme d'analyse syntaxique est finalisé. Passage des données: Le passage des données lexicales aux données syntaxiques n'est pas encore terminé. Matrice de passage: Arnaud a terminé la création de la matrice de passage. Arbre syntaxique: PYJ a finalisé l'arborescence, à l'exception du nœud IDF qui pose des problèmes avec les fonctions et les variables. Programmes Ada: Titouan a réalisé les programmes en Ada, mais quelques modifications sont nécessaires concernant les chevrons.

Disponibilités :
PYJ: Disponible du 23 au 27 décembre. Prochaine réunion Date: 27 décembre Heure: 18h

Tâches à faire :
Pierre: Finaliser l'analyseur syntaxique. Arnaud: Finaliser la deuxième matrice de passage. PYJ: Déboguer le nœud IDF. Titouan: Apporter les modifications nécessaires aux programmes Ada.

VII.11 - CR 11

Compte-rendu - 27 décembre 2023 Résumé: Réunion

Participants: Tout le monde

Date: 2023-12-27

Lieu: Discord

Heure: 18h

Activités:
Analyseur syntaxique: Pierre a finalisé l'analyseur syntaxique, mais l'implémentation du lexeur reste à faire. L'analyseur syntaxique fonctionne en recevant une grammaire sous forme de fichier CSV, similaire au tableau de Gramophone. Le fichier CSV ne doit pas contenir d'espaces. Tous les non-terminaux doivent commencer par « £ ». Une règle vide doit être symbolisée par « § ». Dans une règle, les terminaux/non-terminaux multiples doivent être séparés par « & ». Les epsilons sont symbolisés par « € ». Les terminaux et les non-terminaux ne doivent pas utiliser les symboles suivants autres que pour l'usage décrit ci-dessus : « , » « & » « | » « § » « € » « £ » Le premier non-terminal dans Gramophone est celui qui doit engendrer la grammaire. Le dernier terminal dans Gramophone doit être le terminal qui termine la grammaire (soit « $ »).

Tâches à venir:
Transformer le format de sortie de Gramophone en un CSV lisible par le programme. Finaliser le lexeur en intégrant le nouvel analyseur syntaxique. Établir la liaison entre la grammaire et l'AST : Déterminer le nœud à ajouter ou à modifier lors de la lecture d'un terminal ou d'un non-terminal. Créer un dictionnaire pour traduire ces éléments.

VII.12 - CR 12

Compte-rendu - 3 janvier 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-01-03

Lieu: Discord

Heure: 17h

Activités:
Grammaire: Arnaud a souligné la difficulté de modifier la grammaire tout en conservant sa nature LL(1). Pierre-Yves a confirmé la complexité de la tâche et a proposé de reporter la discussion. Parseur de Gramophone: Titouan a terminé le développement du parseur de Gramophone, qui fonctionne correctement. Le parseur utilise le fichier HTML de Gramophone pour générer un fichier CSV correspondant à la grammaire. Analyse lexicale et syntaxique: Pierre a finalisé le second analyseur lexical et syntaxique.

Tâches à venir:
Création de l'AST (arbre syntaxique abstrait) et détermination de la responsabilité de sa gestion. Mise en place d'une structure permettant d'éliminer les singletons et d'améliorer la lisibilité de l'arbre syntaxique. Vérification de la cohérence entre l'AST et l'arbre syntaxique. Action complémentaire: Approfondir les connaissances sur l'AST et la grammaire.

VII.13 - CR 13

Compte-rendu - 26 janvier 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-01-26

Lieu: 1.8

Heure: 14h

Activités:
Présentation du code réalisé.

Tâches à venir:
Pierre: Implémentation de la table des symboles (TDS). PYJ: Ajout des contrôles sémantiques dans l'AST (arbre syntaxique abstrait). Réflexion commune: Étude des mécanismes de traduction des instructions en langage ARM32. Développement d'une stratégie de navigation dans l'AST pour la traduction.

VII.14 - CR 14

Compte-rendu - 14 février 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-02-14

Lieu: Tek

Heure: 14h20

Activités:
Titouan: Recherche sur les méthodes de génération de code, documentées dans le dépôt Git. Analyse des options d'utilisation de Visual Studio ou d'un Raspberry Pi 64 bits. Arnaud: Réalisation de contrôles sémantiques élémentaires. Pierre: Tâche en cours : Finaliser la table des symboles (TDS) d'ici la semaine prochaine.

Tâches à venir:
Pierre: Finalisation de la TDS.

VII.15 - CR 15

Compte-rendu - 22 février 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-02-22

Lieu: Tek

Heure: 12h40

Activités :
Titouan: Développement d'un outil de génération de code. Implémentation d'une première table des symboles (TDS) avec des fonctionnalités sémantiques (en cours de finalisation). Création d'une deuxième TDS dynamique pour la génération de code. Préparation d'exemples pour illustrer les concepts d'accès et de record. Réalisation d'exemples de génération d'assembleur. Pierre: Travaux en cours sur la finalisation de la sémantique. PYJ: Tâche à venir : Création de la fonction « produire ». Arnaud: Tâche à venir : Mise en place de la TDS pour la génération de code.

Tâches à venir :
Pierre: Finalisation de la sémantique. PYJ: Création de la fonction « produire ». Titouan: Génération de code et parcours de l'arbre. Arnaud: Implémentation de la TDS pour la génération de code.

VII.16 - CR 16

Compte-rendu - 28 février 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-02-28

Lieu: Discord

Heure: 18h00

Activités:
Pierre: Finalisation de la table des symboles (TDS) et de sa sémantique. PYJ: Intégration de la fonction « produire() » dans l'interface. Identification d'un problème avec la boucle for, nécessitant soit l'utilisation d'une pile de registres, soit la création d'une TDS dédiée. Proposition d'une approche plus efficace utilisant une deuxième TDS, simulant l'appel de fonction avec la variable i. Mise en place d'un mécanisme de hachage de l'identifiant de l'objet Java pour générer le nom du saut. Réflexion collective: Analyse du problème des variables globales, concluant à leur absence dans le contexte actuel.

Tâches à venir:
Pierre: Finalisation de la sémantique. Équipe: Création de la TDS pour la génération de code.

VII.17 - CR 17

Compte-rendu - 13 mars 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-03-13

Lieu: Discord

Heure: 18h00

Activités:
Pierre: Poursuite du développement de la deuxième table des symboles (TDS). PYJ: Validation de la première TDS. Consignes générales: Éviter l'utilisation de la fonction put tant qu'elle n'est pas implémentée. Utiliser le compilateur gcc à la place de as en raison de l'utilisation de la fonction fprintf.

Tâches à venir:
Pierre: Lecture du cours sur la traduction et poursuite du développement de la deuxième TDS. PYJ: Prise de mesures pour empêcher les plantages de la traduction et intégration de la fonction put dans les contrôles. Arnaud et Titouan: Test du code de PYJ.

VII.18 - CR 18

Compte-rendu - 21 mars 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-03-21

Lieu: Tek

Heure: 12h30

Activités :
Pierre: Finalisation de la table des symboles (TDS). Tâche à venir : Mise en place des contrôles sémantiques. PYJ: Gestion de la fonction put et traitement d'autres erreurs. Arnaud: Réalisation de tests initiaux. Tâche collective: Ajout de « setters intelligents ».

Tâches à venir :
Titouan: Développement de la génération de code. Pierre: Finalisation de la répartition des variables et des fonctions dans la TDS. Arnaud et PYJ: Débogage et correction des erreurs dans le code.

VII.19 - CR 19

Compte-rendu - 29 mars 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-03-29

Lieu: Tek

Heure: 12h00

Activités:
Arnaud: Exécution de tests. PYJ: Tests effectués, mais échec du code ultérieurement. Pierre: Reprise du travail sur la table des symboles (TDS). Titouan: Début du travail sur l'affectation et les appels de fonction. Mise en place d'un mécanisme de recherche des variables pour l'affectation. Nécessité d'une pile de registres pour l'affectation. Développement initial de la pile de registres sans plantage du code. Nécessité de vérification de la pile de registres. Arnaud: Réalisation de la fonction « loop ».

Tâche à venir :
Création d'une base de données de programmes simples pour des tests plus poussés. Tâches à venir (non exhaustif) Poursuite du développement de la génération de code (Titouan) Finalisation de la TDS et intégration des contrôles sémantiques (Pierre) Tests approfondis du code avec la base de données de programmes (Arnaud) Débogage et correction des erreurs persistantes (équipe)

VII.20 - CR 20

Compte-rendu - 2 avril 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-04-02

Lieu: Tek

Heure: 13h20

Activités:
Arnaud: Tests des conditions logiques (vrai/faux). PYJ: Collaboration avec Moufida pour la suite du travail. Pierre: Finalisation des modifications liées aux variables. Titouan: Corrections apportées au code d'Arnaud. Nécessité de validation par Pauline concernant l'instanciation des variables. Abandon de l'utilisation de la pile des registres au profit d'une pile standard.

Tâches à venir:
Vérification des erreurs sémantiques (priorité). Poursuite du développement de la génération de code.

VII.21 - CR 21

Compte-rendu - 7 avril 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-04-07

Lieu: Discord

Heure: 17h00

Activités:
Pierre: Finalisation de la table des symboles (TDS) pour les noms de variables, en tenant compte de la gestion de deux variables portant le même nom dans la même TDS. Tâches en cours: Création de la TDS pour les noms de fonctions et procédures. Prise en charge des aspects sémantiques avant la traduction du code. Arnaud: Tests des programmes pour identifier les erreurs syntaxiques et de génération de code. Développement d'un outil permettant la compilation de tous les programmes.

Tâches à venir:
PYJ: Mise en place du traitement des variables « for ». Pierre: Poursuite du travail sur la TDS des fonctions. Titouan: Poursuite du développement des fonctionnalités liées aux fonctions. Arnaud: Continuation des tests et amélioration de l'outil de compilation.

VII.22 - CR 22

Compte-rendu - 18 avril 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-04-18

Lieu: Anim'Est

Heure: 13h00

Activités:
Titouan: Tâche à venir : Fixation des fonctions pendant les vacances. PYJ: Identification d'un problème de TDS null dans la boucle for, lié à la variable tds_actuelle. Tâches collectives: Réalisation d'un programme illustrant le problème de TDS null pour la soutenance. Développement initial du programme en Python. Titouan exprime son souhait de travailler sur les enregistrements.

Tâches à venir:
Arnaud: Prise en charge du développement du programme pour illustrer le problème de TDS null. Tous: Continuation des tâches individuelles en cours.

VII.23 - CR 23

Compte-rendu - 4 mai 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-05-04

Lieu: Discord

Heure: 11h00

Activités:
Arnaud: Recherche de petits programmes à tester et questionnement sur l'exhaustivité des tests nécessaires. Discussion collective: Décision de ne pas implémenter de tableaux ni de listes pour des raisons de temps. Choix d'implémenter les fonctions Fibonacci et Tribonacci dans une même fonction différenciée. Nécessité d'utiliser divers éléments du langage dans le programme de test: Fonctions et procédures Variables portant le même nom Fonctions imbriquées Boucles for et if Fonctions récursives Opérations avec ordre d'exécution Débogage: Résolution de problèmes liés au « bus de la génération ». Tests des variables globales après correction du « bus de la génération ». Passage aux tests des procédures.

Tâches à venir:
Vérification de la précédence de création des fonctions et procédures. Ajout d'AppelProcédure au PlantUML. Tests approfondis des procédures et fonctions, notamment les appels. Gestion des erreurs sémantiques pour éviter l'arrêt du programme.

Prochaine réunion:
Date: Mercredi 8 mai 2024 Heure: 18h00

VII.24 - CR 24

Compte-rendu - 8 mai 2024 Résumé: Réunion

Participants: Tout le monde

Date: 2024-05-08

Lieu: Discord

Heure: 18h00

Activités:
Avancement des tâches individuelles: Les participants ont poursuivi le travail sur leurs tâches respectives. Problèmes rencontrés: Persistance de problèmes liés aux déplacements de variables. Découverte d'un bug lors de l'utilisation de 260 variables dans une même fonction. Tâches à venir Priorités: Finalisation de la génération des procédures et tests associés. Vérification de la précédence des fonctions et procédures. Ajout d'AppelProcédure au PlantUML. Tests approfondis des procédures et fonctions, notamment les appels. Gestion des erreurs sémantiques pour éviter l'arrêt du programme. Tâches complémentaires: Création d'un compilateur sans tous les « system out ». Implémentation des instructions « and then » et « or else ». Suppression des instructions « in » et « out ».

Répartition des tâches écrites :
Pierre: Partie sur la TDS Pauline & Titouan: Génération de code Arnaud: Gestion de projet et programme