Compte rendu de projet

PCL 2 - compte rendu final

Marine RECHER
Antoine YEBOUET
Louis LAFOND

Année 2022–2023

Professeurs : COLLIN S., DA SILVA S.

Rendu le mercredi 12 avril 2023.

Table des matières

1 Introduction

Ce rapport de fin de projet PCL complète nos rapports d'activités de PCL1, et synthétise le travail effectué pendant la seconde partie du projet, la partie PCL2.

Commençons par une petite remise en contexte : pendant la première partie du projet (PCL1), nous avons commencé par écrire la grammaire du langage Tiger. Cette grammaire, nous l'avons testée sur des exemples variés de programmes plus ou moins complexes, avec et sans erreurs lexicales et syntaxiques (voir rapport n°1 PCL1). Ensuite survient le développement de l'arbre abstrait, une étape clef dans ce projet puisque nous en aurons besoin régulièrement pour la suite. On définit les analyses lexicales et sémantiques en parallèle, ainsi que la mise en place de l'arbre abstrait qui doit permettre de représenter graphiquement le parcours d'une instruction Tiger par notre compilateur (voir rapport n°2 PCL1).

Une fois l'AST construit, nous nous sommes penchés sur la table des symboles. Nous avons ensuite implémentés tous les contrôles sémantiques liés au langage Tiger. Lors de la soutenance finale de la partie PCL1, en janvier 2023, nous avons présenté l'arbre abstrait, la table des symboles ainsi que l'ensemble des contrôles sémantiques implémentés (voir rapport n°3 PCL1). Lors de cette présentation, notre analyse sémantique était encore imparfaite, un dernier problème subsistait, que nous n'étions pas arrivés à régler dans les temps : l'appel de fonction récursif. Aussi, nos tables des symboles ne prenaient pas en compte les déplacement. Ce sont donc les deux choses que nous avons faites pendant le module PCL2, qui a débuté en janvier et doit se terminer après la soutenance finale, le jeudi 13 avril 2023.

2 Notre production PCL2

2.1 Production du travail

Pour décrire la seconde partie du travail (concernant PCL2 uniquement et la production de code assembleur), nous avons procédé comme suit : nous nous sommes d'abord concentré sur la compréhension du code assembleur à écrire pour une fonction "type". Nous avons pris quelques heures pour définir certaines conventions pour pouvoir se répartir les fonctions entre nous.

Nous avons commencé par comprendre que nous allions utiliser l'"AstVisitor" pour parcourir les noeuds de notre AST créé et générer le code assembleur en conséquence, en utilisant les tables des symboles mises en place lors du premier parcours.

Pour générer du code assembleur, nous avons décidé de retranscrire des chaînes de caractères dans un fichier à part (notre fichier " \out\main.s ") avec l'aide de notre fonction "write()". Lors du parcours de l'AST, chaque instruction ets donc traduite en ARM et le code ARM résultant est retranscrit, ligne par ligne, dans un fichier particulier. Ce fichier est directement exécutable avec l'outil "Visual" qui nous permet de simuler l'exécution pas à pas de notre code ARM, et de visualiser la pile et les registres correspondant.

Chaque noeud a ses spécificités, pour autant, nous n'allons pas détailler toutes les méthodes. Dans la partie qui suit, nous avons choisi de vous présenter celles qui nous semblent les plus pertinentes.

Utilisation des registres :

- R0 : Résultats d'opérations et retours de fonctions
- R1, R2 : Opérateurs
- R3 : Compteur chaînage statique
- R2, R4 : Bornes boucles for
- R10 : Changement de base lors du chaînage statique
- R11 : Registre de base
- R13 : Sommet de pile

2.2 Schémas de traduction du langage TIGER

2.2.1 Les opérateurs de base

Les opérateurs de base " =, <>, <, >=, >, <=, |, , +, -, * et /" se traduisent presque tous de manière transparente en utilisant les méthodes de base en ARM (par exemple avec ADD pour traduire l'addition +). Ce n'est pour autant pas le cas des opérateurs de multiplication et de division, qui nécessitent plus de recherche.

Ci-dessous l'écriture de la fonction "Addition" ainsi que la fonction "<=".

```
@Override
public String visit(Plus plus) {
    write(s:";    addition");
    plus.left.accept(this);
    write(s:"\tMOV R1,R0");
    plus.rigth.accept(this);
    write(s:"\tADD R0,R1,R0");
    return null;
}
```

```
1  ;    addition
2      LDR R0,=1
3      MOV R1,R0
4      LDR R0,=2
5      ADD R0,R1,R0
6      END
```

FIGURE 2.1 – Une fonction "addition" et le code ARM résultant pour le code tiger : **1+2**

On place la valeur de gauche dans R1 et la valeur de droite se trouve dans R0, donc il suffit d'additionner les deux registres et stocker le résultat dans R0.

```
@Override
public String visit(LessThan1 lessThan1) {
    write(s:";    <=");
    lessThan1.left.accept(this);
    write(s:"\tSTR R0,[R13,#-4]!");
    lessThan1.rigth.accept(this);
    write(s:"\tLDMFD R13!,{R1}");
    write(s:"\tMOV R2,R0");
    write(s:"\tMOV R0,#0");
    write(s:"\tCMP R1,R2");
    write(s:"\tMOVLE R0,#1");
    return null;

}
```

```
;    <=
    LDR R0,=1
    STR R0,[R13,#-4]!
    LDR R0,=2
    LDMFD R13!,{R1}
    MOV R2,R0
    MOV R0,#0
    CMP R1,R2
    MOVLE R0,#1
    END
```

FIGURE 2.2 – Une fonction "<=" et le code ARM résultant pour le code tiger **1 <=2**

On place la valeur de gauche dans la pile. Cela permet d'éviter que la valeur de gauche soit effacé par une nouvelle comparaison dans la valeur de droite. On place la valeur de droite dans R2 et on récupère de la pile la valeur de gauche qu'on place dans R1. On compare les deux registres et on ne modifie R0 à 1 seulement si R1 est plus petit que R2.

2.2.2 L'accès à une variable

```
;    début programme: let in end
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!
;    déclaration de fonction
    B func_end_0     ;on saute la fonction si elle n'est pas appelée
func_func
    STMFD R13!,{LR}   ;adresse de retour
;    multiplication
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    MOV R1,R0
;    identifier
    MOV R3,#1    ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_1
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_1    ; si on a remonté tout le chaînage statique
    LDR R10,[R10]
    B loop_chainage_statique_1
exit_chainage_statique_1
    LDR R0,[R10,#-4]
    MOV R2,R0
    BL mul     ;on branche vers le code commun pour toutes les multiplications
    LDMFD R13!,{PC}    ;on revient à l'adresse de retour
func_end_0
;    appel de fonction
    STR R11,[R13,#-4]!
    MOV R11,R13
    LDR R0,=2
    STR R0,[R13,#-4]!     ;on empile tous les paramètres
    BL func_func     ;on se déplace au code de la fonction (et dans la TDS de la fonction)
    LDMFD R13!,{R12}      ;on dépile les paramètres
    LDMFD R13!,{R11}
prog_end ;program end
    END
```

```
1   let
2       var a := 0
3
4       function func(n: int) =(
5           n * a
6       )
7
8   in
9       func(2)
10  end
```

FIGURE 2.3 – Une fonction de test d'accès à une variable et le code ARM résultant

Variable locale

Si la variable est locale, il suffit de se déplacer dans la pile à Base + Déplacement et de récupérer la valeur associée à cette adresse.

Variable non locale

Si la variable n'est pas locale, on utilise le chaînage statique pour se placer dans la bonne base. Pour savoir le nombre de déplacements à faire, on utilise la valeur Nx-Ny, avec Nx le numéro d'imbrication du bloc courant, et Ny celui du bloc dans lequel la variable a été définie. Puis on se déplace à Base + Déplacement comment dans le cas précédent. (voir la figure en début de paragraphe)

2.2.3 Appel de fonction

On enregistre dans la pile l'adresse de l'ancienne base, puis on met à jour la base. S'il y a des paramètres, on les empile. On branche vers le label du corps de la fonction où se trouvent les instructions à effectuer. Une fois la fonction terminée, on dépile les paramètres qu'on avait empilés et on retourne à l'ancienne base.

```
;    début programme: let in end
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!
;    déclaration de fonction
    B func_end_0     ;on saute la fonction si elle n'est pas appelée
func_myfunc
    STMFD R13!,{LR}   ;adresse de retour
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
;    addition
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    MOV R1,R0
    LDR R0,=1
    ADD R0,R1,R0
;    assignement
    STR R0,[R11,#-4]       ;si la variable est locale
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    LDMFD R13!,{PC}    ;on revient à l'adresse de retour
func_end_0
;    identifier
    MOV R3,#1    ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_1
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_1    ; si on a remonté tout le chaînage statique
    LDR R10,[R10]
    B loop_chainage_statique_1
exit_chainage_statique_1
    LDR R0,[R10,#-4]
;    appel de fonction
    STR R11,[R13,#-4]!
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!     ;on empile tous les paramètres
    BL func_myfunc     ;on se déplace au code de la fonction (et dans la TDS de la fonction)
    LDMFD R13!,{R12}      ;on dépile les paramètres
    LDMFD R13!,{R11}
;    assignement
    STR R0,[R11,#-4]       ;si la variable est locale
prog_end ;program end
    END
```

```
1  let
2      var a :=0
3      function myfunc(x:int):int = (
4          x:=x+1;
5          x
6      )
7  in
8      a:= myfunc(0)
9  end
```

FIGURE 2.4 – une fonction de test d'appel de fonction et le code ARM résultant

2.2.4 Les structures de contrôle (while)

On écrit le label while en ajoutant un id qui s'incrémente à chaque label utilisé. L'id permet de différencier les différentes boucles while entre elles. Pour plus de simplicité, on a décidé d'utiliser

une seule variable commune à tous les labels, il est donc impossible d'avoir deux labels identiques dans le code assembleur, ce qui pourrait provoquer une erreur lors de l'exécution.

Après avoir visité la condition du while, on compare le résultat de celle-ci contenu dans le registre 0 pour savoir si elle est vérifié ou pas. Si le registre 0 est égale à 0, c'est que la condition du while n'est pas vérifiée et on fait un branchement pour aller jusqu'au label qui indique la fin de la boucle while. Sinon, on visite le bloc d'instruction contenu dans le bloc while. Une fois celui-ci termniné, on fait un branchement pour retourner au début de la boucle.

(Problème avec Latex, voir les figures plus bas.)

```
;    début programme: let in end
    MOV R11,R13
    LDR R0,=8
    STR R0,[R13,#-4]!
;    boucle while
;    >
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    STR R0,[R13,#-4]!
    LDR R0,=0
    LDMFD R13!,{R1}
    MOV R2,R0
    MOV R0,#0    ;on initialise le résultat à FALSE
    CMP R1,R2      ;si valeur de gauche > valeur de droite, R0=1
    MOVGT R0,#1
    STR R0,[R13,#-4]!    ;on empile la valeur dgge la condition, pour les imbrications
while_0
;    >
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    STR R0,[R13,#-4]!
    LDR R0,=0
    LDMFD R13!,{R1}
    MOV R2,R0
    MOV R0,#0    ;on initialise le résultat à FALSE
    CMP R1,R2      ;si valeur de gauche > valeur de droite, R0=1
    MOVGT R0,#1
    CMP R0,#0      ;on compare la valeur de la condition, si faux on branche à la fin
    BEQ end_while_0
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
;    soustraction
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    MOV R1,R0
    LDR R0,=1
    SUB R0,R1,R0
;    assignement
    MOV R3,#2   ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_1
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_1
    LDR R10,[R10]
    B loop_chainage_statique_1
exit_chainage_statique_1
    STR R0,[R10,#-4]       ;on met 'au bon endroit' la valeur de R0 (l'affectation)
    B while_0
end_while_0
    LDMFD R13!,{R0}
prog_end ;program end
    END
```

```
1  let
2      var N := 8
3  in
4      while N > 0 do
5          ( N:=N-1 )
6
7  end
```

FIGURE 2.5 – Une fonction de test "while" et le code ARM résultant

2.2.5 Les structures de contrôle imbriquées (for imbriqué)

```
;    début programme: let in end
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!
;    boucle for
    STMFD R13!,{R2,R4}     ;on empile les potentielles bornes du for précedent (s'il existe)
    STR R11,[R13,#-4]!     ;on empile l'ancienne base avant de la maj,chainage statique
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!
    MOV R4,R0     ;R4 borne minimale
    LDR R0,=3
    MOV R2,R0     ;R2 borne maximale
for_0
    CMP R4,R2      ;si min > max arret de la boucle
    BGT end_for_0
;    boucle for
    STMFD R13!,{R2,R4}     ;on empile les potentielles bornes du for précedent (s'il existe)
    STR R11,[R13,#-4]!     ;on empile l'ancienne base avant de la maj,chainage statique
    MOV R11,R13
    LDR R0,=0
    STR R0,[R13,#-4]!
    MOV R4,R0     ;R4 borne minimale
    LDR R0,=3
    MOV R2,R0     ;R2 borne maximale
for_1
    CMP R4,R2      ;si min > max arret de la boucle
    BGT end_for_1
;    identifier
    MOV R3,#2    ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_2
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_2    ; si on a remonté tout le chaînage statique
    LDR R10,[R10]
    B loop_chainage_statique_2
exit_chainage_statique_2
    LDR R0,[R10,#-4]
;    addition
;    identifier
    MOV R3,#1    ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_3
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_3    ; si on a remonté tout le chaînage statique
    LDR R10,[R10]
    B loop_chainage_statique_3
exit_chainage_statique_3
    LDR R0,[R10,#-4]
    MOV R1,R0
;    identifier
    LDR R0,[R11,#-4]       ;si la variable est locale, on la met dans R0
    ADD R0,R1,R0
;    assignement
    MOV R3,#2   ;sinon, on utilise le chaînage statique (R3 contient NX-NY)
    LDR R10,[R11]
loop_chainage_statique_4
    SUBS R3,R3,#1
    BEQ exit_chainage_statique_4
    LDR R10,[R10]
    B loop_chainage_statique_4
exit_chainage_statique_4
    STR R0,[R10,#-4]       ;on met 'au bon endroit' la valeur de R0 (l'affectation)
    ADD R4,R4,#1       ;incrémenter min
    STR R4,[R11,#-4]
    B for_1
end_for_1
;    dépiler le compteur et la base de la TDS du for
    LDMFD R13!,{R0,R11}
    LDMFD R13!,{R2,R4}
    ADD R4,R4,#1       ;incrémenter min
    STR R4,[R11,#-4]
    B for_0
end_for_0
;    dépiler le compteur et la base de la TDS du for
    LDMFD R13!,{R0,R11}
    LDMFD R13!,{R2,R4}
prog_end ;program end
    END
```

```
1   let
2       var a := 0
3   in
4
5       for i := 0 to 3 do (
6           for j := 0 to 3 do (
7               a := i + j
8           )
9       )
10  end
```

FIGURE 2.6 – Une fonction de test "for imbriqué" et le code ARM résultant

On utilise R2 et R4 pour les bornes du for, faire un bloc for dans un bloc for effacera les valeurs des bornes du premier for, car on réécrit par-dessus dans les registres. Il faut donc stocker les valeurs de R2 et de R4 dans la pile lors de l'entrée d'une boucle for.

On enregistre l'adresse de l'ancienne base dans la pile pour le chaînage statique et on met à jour la base. Puis, on initialise les bornes minimales et maximales dans R4 et R2. On compare R4 et R2, si la borne min est égale à la borne max alors on branche à la fin de la boucle for. On fait les instructions dans le corps de la boucle for et on incrémente la borne minimale dans R4.

A la fin du for, on récupère l'ancienne base et les anciennes bornes dans R2 et R4.

3 Gestion de projet

Nous étions 4 étudiants au début de ce projet, mais nous avons continué notre travail lors du second semestre (PCL2) à 3. Nous avons donc dû nous organiser en conséquence.

Comme précisé précédemment, au commencement de PCL2, il nous restait deux choses à finir pour pouvoir écrire la génération de code assembleur proprement : l'appel de fonctions récursives, qui nous posait problème à la fin de PCL1, et les déplacements. Nous avons eu besoin de quelques séances pour terminer de produire ces fonctions essentielles pour la suite. Ci-dessous, un tableau récapitulant le temps total approximatif passé sur ces trois activités.

tâche	durée
correction de l'analyse sémantique	**15**
ajout du déplacement à nos TDS	**20**
génération de code ARM	**85**

FIGURE 3.1 – Répartition du travail, en heures

Au commencement de la production de code assembleur, nous avons écrit quelques fonctions ensemble, les fonctions que nous estimions les plus importantes pour faire la suite. Nous avons donc commencé par écrire tous les trois les fonctions concernant la visite des noeuds "Or", "And" ainsi que "Addition". Cette première approche nous a permis de nous familiariser avec le langage ARM en réfléchissant tous les 3 au développement de la syntaxe. Elle nous a aussi permis d'apprendre à utiliser le logiciel Visual.

A propos de la méthode de travail, nous nous réunions toutes les semaines pour travailler "en présentiel" à l'école avec un liveshare, pour pouvoir solliciter l'aide des autres membres du groupe en cas de questionnement. Lorsque certaines tâches nous ont semblées assez faciles pour être réalisées seules de notre coté, on se les répartissait. Mais la plupart du temps, nous avons travaillé tous les trois en simultané, pour ne pas être bloqués en cas de problème et faire vérifier notre code directement entre nous. Nous avons donc avancé pas à pas dans le développement de notre compilateur, en commençant par les tâches qui nous semblaient les plus importantes et les plus faciles à réaliser pour obtenir un compilateur fonctionnel.

Ci-dessous, un tableau récapitulant la répartition progressive de notre travail de génération de code assembleur :

		niveau	1	2	3
	Travail fixé pour la séance				
Louis :	opérateur logique (& , | , +, -)	expression arithmétique (<, <=)	structure de contrôle (if, for)	définition de fonction	
Marine :	opérateur logique (& , | , +, -)	expression arithmétique (>, >=)	structure de contrôle (while)	appel de fonctions	
Antoine :	opérateur logique (& , | , +, -), opérateur (*, /)	expression arithmétique (=, <>), affectations simples	structure de contrôle (if, for)	déclaration de variables, appel de fonctions	if, while,for imbriqués, break

FIGURE 3.2 – Répartition des tâches, et leurs niveaux

4 Annexe

Le code source de notre programme de démonstration :

```
1   let
2       var a := 0
3
4       function carre(n: int) =(
5           n * n
6       )
7
8   in
9       while a<5 do (
10          a := a + 1
11      );
12
13      for i := 0 to 3 do (
14          for j := 0 to 3 do (
15              a := i + j
16          )
17      );
18
19      if a > 3 then (
20          a := 6
21      )else(
22          a := 7
23      );
24
25      a := carre(a)
26  end
```

FIGURE 4.1 – Notre programme de démonstration, 'all.tiger'