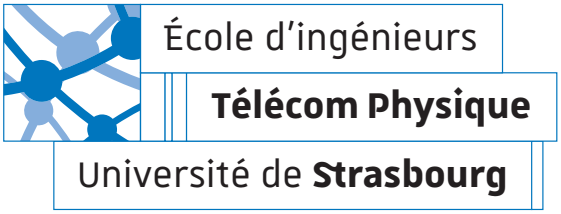

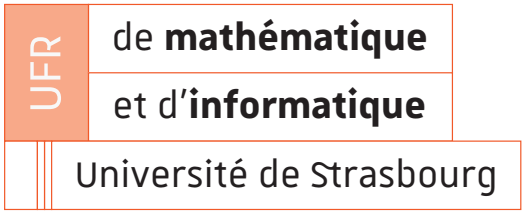

Ricoré®, un compilateur DECAF pas comme les autres

Projet de Compilation

Erwan Escooldie David Nicolazo Hugo Castel Théo Bastien
Vagnona Andrianandrasana-Dina

Année 2021–2022

Résumé

Ricoré® est un compilateur du langage minimaliste DECAF supportant l'ensemble des fonctionnalités de ce dernier. Ce rapport décrit les différents choix d'implémentation, les problématiques rencontrées et solutions mises en place.

Table des matières

Table des matières **i**

1 Introduction **1**

2 Contraintes et solutions **2**
2.1 Types . . . 2
2.2 Portée des variables . . . 2
2.3 Emplacements . . . 2
2.4 Affectations . . . 3
2.5 Invocation de méthode et retour . . . 3
Paramètres . . . 3
Appel . . . 3
Retour . . . 4
2.6 Structure de contrôle . . . 4
`if` . . . 4
`for` . . . 4
2.7 Expressions . . . 4
2.8 Inclusion de routine de bibliothèque externe . . . 4

3 Liste des quadruplets **5**

4 Conclusion **7**

1 Introduction

La compilation est une étape essentielle dans la conception d'un programme car elle consiste ni plus ni moins à traduire en langage machine un programme écrit en un langage lisible par l'homme.

En effet l'assembleur n'étant qu'une translitération des instructions binaires effectivement utilisées par les processeurs, le compilateur est souvent la dernière étape dans laquelle notre programme change significativement de forme. Tous les concepts qui peuvent parfois rendre la programmation ludique aux plus novices comme les objets, les structures de contrôle, la gestion dynamique des erreurs disparaissent pour ne laisser place qu'à des instructions élémentaires se satisfaisant bien de piles et d'adresses de branchement.

Le compilateur a donc à charge de garder l'intégrité du programme lors de la traduction, mais aussi de s'assurer qu'il propose une version optimisée de ce dernier (le programmeur n'ayant en général pas accès aux choix fait par le compilateur depuis le langage haut niveau).

Ce projet de compilation est donc une très bonne occasion non seulement de mettre en pratique les notions vues en cours de Compilation, mais aussi de réellement faire face aux problématiques rencontrées par ces programmes clés de voûte.

Nous rappellerons ici les différents attendus du sujet en présentant brièvement nos mises en œuvre pour les résoudre, puis nous exposerons la liste des opérateurs utilisés pour les quadruplets de notre langage intermédiaire, avant d'enfin conclure.

2 Contraintes et solutions

2.1 Types

Nous avons implémenté 5 types : `int`, `boolean`, ainsi que leurs versions en tableaux `int[]` et `boolean[]`, ainsi que le type `string` qui n'est supporté que dans la fonction *native* `WriteString`. `char` n'est qu'un alias de `int` car passé l'étape de l'analyse lexicale et syntaxique, nous avons fait le choix de la traiter comme tel.

De par sa nature un peu à part, le traitement des `string` s'est révélé un peu plus fastidieux que prévu car il ne s'agit finalement pas d'un type à part entière car il n'est utilisable que via la fonction `WriteString`. Il bénéficie donc d'un traitement à part dans notre compilateur.

2.2 Portée des variables

Dans tous les contextes, la syntaxe DECAF impose de déclarer l'ensemble de nos variables avant de les exploiter, et ce à tout les niveaux de portée.

On dénombre 3 niveaux de portée :

- Globale, déclarées au niveau de la `class` et accessible partout,
- Méthodes, déclarées au début d'une méthode et accessible dans celle-ci,
- Locales, déclarées au début d'un *bloc* interne à une fonction, accessible uniquement depuis ledit bloc.

De cette manière, il est possible de faire cohabiter des variables portant le même nom tant qu'elles se trouvent dans des portées distinctes. Lors de la résolution d'un nom de variable, on cherche de la portée la "plus locale" à la "plus globale" (voir fig. 2.1). Il n'y a donc de conflits que si l'on tente de créer deux variables portant le même nom dans un même contexte.

Pour mettre en œuvre ce mécanisme, nous avons mis en place des *tables de contexte* et un *arbre des contextes* ce qui nous a permis, à partir d'un *contexte courant*, de remonter jusqu'au contexte global en passant par les contextes intermédiaires encapsulant notre contexte courant. Cette remontée est effectuée à chaque modification de variable afin de déterminer quelle variable le programmeur veut affecter. La table du contexte courant est également parcourue à chaque déclaration de variable dans ce contexte afin de détecter les éventuels conflits.

2.3 Emplacements

À l'initialisation, toutes les variables sont initialisées à une valeur par défaut (`0` pour les `int`, `false` pour les `boolean`). Le choix a été fait d'allouer une zone statique pour les variables du contexte globale. En ce qui concerne les variables de méthodes et autres locales, ces dernières sont directement allouées sur la pile car les tableaux ne sont pas autorisés dans ces niveaux.

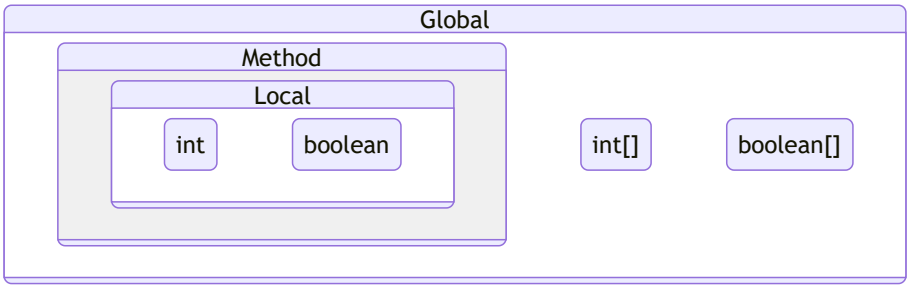

Figure 2.1: Visualisation des encapsulations de contextes

2.4 Affectations

Les affectations se font de manière générale par l'intermédiaire du symbole `=` et accessoirement pour les `int` via les raccourcis `+=` et `-=`.

Les tableaux en eux-mêmes sont inexploitables et doivent systématiquement être décorés d'un *index* après leur déclaration afin de se rapporter à l'usage d'un `int` ou d'un `boolean`.

Les arguments des méthodes sont passés par valeurs et non pas par référence : les variables données en arguments de méthode sont copiés telles des nouvelles variables dans le contexte de méthode, il est donc possible de les réaffecter et de travailler avec sans que cela ne se répercute sur les variables de l'appelant. Cela permettait de généraliser le comportement dans le cas où une méthode reçoit des arguments variables ou des arguments littéraux.

2.5 Invocation de méthode et retour

D'après le sujet,

- une méthode quelconque peut être appelée dans `statement`
- dans `expr`, seule une méthode renvoyant un résultat être appelée
- une méthode ne peut renvoyer qu'un `int` ou `boolean`
- les arguments sont placés sur la piles juste avant l'appel
- l'appelé sauvegarde `$fp` et `$ra` sur la pile et execute son code
- lorsqu'un `return` apparaît, il est passé de bloc en bloc jusqu'a atteindre le contexte méthode (avec un marqueur `method`), l'appelant le traitera.

Paramètres

Les paramètres de la fonction après avoir été calculés (si expression complexe), sont placés sur la pile de gauche à droite, à l'aide d'un quadruplet

Appel

La méthode est ensuite appelée par le quadruplet `Q_CALL_METH`, plaçant son `$fp` juste avant les arguments et sauvegardant `$fp` et `$ra` sur la pile avant d'exécuter son code.

Retour

Si la méthode ne renvoi rien, elle peut être appelée depuis un `statement` seulement. Sinon elle peut l'être depuis un `expr` ou un `statement` (son résultat est alors oublié). La rencontre d'une fin textuelle de fonction renvoyant un résultat, est considérée comme une erreur.

2.6 Structure de contrôle

`if`

Cette structure de contrôle n'a pas posé problème et n'a été qu'une simple application de cours.

`for`

Un *index itérateur* a été mis en place, initialisé à la première valeur de l'intervalle `borne_inf` et la boucle est répétée tant que la condition `index` ≤ `borne_sup` est vérifiée en incrémentant `index` entre chaque itération.

Les complexités se sont faites sentir au moment de l'implémentation des mots-clés `break` et `continue` : lorsqu'un de ces derniers est rencontré, un *saut* est les adresses de ces sauts sont alors passées de proche en proche jusqu'à rencontrer un contexte de `for`. Les sauts sont alors complétés :

- à l'instruction d'incrémentation d'`index` pour `continue`,
- à la sortie du `for` pour `break`.

A noter qu'en SPIM, puisque le break et le continue sautent du code, il faut dépiler en conséquence les blocs sautés, puis ensuite coder le saut.

2.7 Expressions

Nous avons délégués la gestion des priorités entre les opérateurs à Yacc via les directives types `%left op` notamment grâce à la grande lisibilité que cette solution offre par rapport à une méthode en cascade.

2.8 Inclusion de routine de bibliothèque externe

L'entête d'une routine est inclus au début de la grammaire dans le contexte global (de sorte qu'on ne puisse redéfinir ces méthodes par la suite). Lors de l'écriture du fichier en assembleur, le corps de la routine y est inscrit, comme une méthode lambda.

Cela concerne les fonctions natives `ReadInt`, `WriteInt`, `WriteString` et `WriteBool`.

3 Liste des quadruplets

`Q_ADD`, `Q_SUB`, `Q_MULT`, `Q_DIV`, `Q_RES` (`Q_OP`, *opérande1*, *opérande2*, *destination*)

opérande1 est sur la pile (étant un `expr int`, donc un temporaire)
opérande2 est sur la pile (étant un `expr int`, donc un temporaire)
destination est sur la pile (étant un `expr int`, donc un temporaire)
Execute l'operation *opérande1 op opérande2* et place le résultat dans *destination*.

`Q_EQ`, `Q_NOT_EQ` (`Q_OP`, *opérande1*, *opérande2*, *destination*)

- **cas opérande `int`:**

  produit un branchement `if` (*op1 op op2*) `goto` *destination*

  opérande1 est sur la pile (étant un `expr int`, donc un temporaire)

  opérande2 est sur la pile (étant un `expr int`, donc un temporaire)

  destination est const (ligne)

- **cas opérande `boolean`:**

  réification des deux opérandes puis idem que pour `int` ↑

`Q_LESS`, `Q_LESS_EQ`, `Q_GREAT`, `Q_GREAT_EQ` (`Q_OP`, *opérande1*, *opérande2*, *destination*)

produit un branchement `if` (*op1 op op2*) `goto` *destination*
opérande1 est sur la pile (étant un `expr int`, donc un temporaire)
opérande2 est sur la pile (étant un `expr int`, donc un temporaire)
destination est const (ligne)

`Q_COPY` (`Q_COPY`, *source*, *déplacement*, *destination*)

copie la valeur de *source* vers *destination*
Types de destinations possibles :

- Variable globale : on cherche alors la valeur dans le segment data
- Tableau : on cherche la valeur dans le segment data à *destination* + 4 ×*déplacement*
- Variable locale : on cherche dans la pile

`Q_IF` (`Q_IF`, *opérande1*, _, *destination*)

produit un branchement `if` (*op1*) `goto` *destination*
opérande1 est sur la pile

`Q_PUSH_CTX` (`Q_PUSH_CTX`, , , _)

on push sur la pile, le premier contexte enfant du contexte courant pas encore exploré

`Q_POP_CTX` (`Q_PUSH_CTX`, , , _)

on pop le contexte courant de la pile, et on retour au contexte parent du contexte courant

`Q_BREAK`, `Q_CONTINUE`, `Q_RETURN` (`Q_KW` , , , _)

pop les contextes entre le contexte contenant l'arret (`break`, `continue` ou `return`) et le contexte cible (premier contexte `for` ou `method` rencontré)

Crée un saut jusqu'à ce contexte cible.

`Q_DEF_METH` (`Q_DEF_METH`, *nom_de_fonction*, *nb_d'argument*, _)

Sauvegarde `$fp` et `$ra` sur la pile et actualise `$fp` avant les arguments

`Q_END_METH` (`Q_END_METH`, *nom_de_fonction*, ,)

si *nom_de_fonction* est vide, écrit un code erreur précisant que la fin textuelle d'une fonction a été atteinte et arrête le code

si *nom_de_fonction* est main, arrête le code

sinon restaure `$fp` et `$ra` et retourne au code (`jr $ra`)

`Q_PARAM` (`Q_PARAM`, *depl_sur_la_pile*, _, *arg*)

dpl sur la pile est nécessaire, il correspond au numero de l'argument (le premier y va avec un *dpl* = 0, le second un *dpl* = 4, le troisieme un *dpl* = 8, etc.)

arg est le nom du temporaire correspondant à l'argument ajouté (réifié si c'est un boolean)

`Q_CALL_METH` (`Q_CALL_METH`, *nom_de_fonction*, *nb_d'argument*, *temporaire_contenant_le_retour*)

branchement vers la méthode

4 Conclusion

Ce projet fût riche de découvertes et mises en pratiques de notions parfois très abstraites vu en cours qui prennent finalement tous leur sens une fois face au problème.

Notre production n'est pas parfaite, et des cas très spécifiques de méthodes surchargées que nous pensions pouvoir supporter n'est finalement pas fonctionnel, mais notre compilateur a selon nous le mérite d'être totalement fonctionnel sur les tests proposés, et constitue donc une base compétitive dans le cadre des compilateurs DECAF.