Rapport du projet de compilateur SCALPA

Danyl El-Kabir
Jérémy Bach
Nadjib Belaribi
François Grabenstaetter

3 janvier 2021

Table des matières

1 Résumé

L'objectif de ce rapport est de résumer les capacités de notre compilateur SCALPA. Le compilateur génère du code MIPS à partir d'un *pseudo code Pascal* appellé SCALPA. Nous avons réalisé ce projet de manière incrémentale, en implémentant les différentes fonctions au fur et à mesure.

2 Développement du compilateur

Comme cité précédemment, le développement de ce compilateur s'est fait de manière incrémentale en augmentant petit à petit les capacités du compilateur (en étendant la grammaire).

Dans un premier temps, nous avons réalisé *l'analyse lexicale* permettant d'analyser un programme SCALPA.

Ensuite, nous avons limité le compilateur aux *opérations arithmétiques* ainsi que les *affectations des variables*. Cette version minimale de la grammaire nous à permis de générer notre premier code MIPS et de pouvoir partir sur de bonnes bases.

Une fois cette première étape complètement terminée, nous avons pu élaborer un **jeu de tests** pour les affectations et les opérations arithmétiques pour enfin ajouter de nouvelles fonctionnalités au compilateur en étendant davantage la grammaire.

C'est ainsi que nous avons ajouté le support des *conditions* avec les variables booléennes. L'ajout de ces conditions nous a permis d'ajouter le support des *boucles* while également.

Par la suite, nous avons étendu la grammaire avec l'ajout du support des tableaux et des fonctions, tout en testant régulièrement nos implémentations.

Après avoir testé le bon fonctionnement du compilateur dans sa plus grande partie, nous avons ajouté un module d'optimisation de code.

3 Spécification complète de notre compilateur SCALPA

Le compilateur est donc capable de générer un programme *MIPS R2000* (utilisable sur un simulateur de processeur comme SPIM ou Mars) à partir d'un programme écrit en SCALPA.

Voici une liste non exhaustive des capacités de notre compilateur SCALPA :

Expressions arithmétiques et affectations		
Opérateur	Support	Commentaire
Déclaration de variables		
var a, b : type	X	Déclaration multiple
Affectations et opérations sur les entiers		
:=	X	Affectation entière
+	X	Addition entière
-	X	Soustraction entière
-(unaire)	X	Moins unaire
*	X	Multiplication
/	X	Division
∧	X	Opérateur exponentiel
%	X	Modulo
< ou >	X	Comparaisons entières strictes
<= ou >=	X	Comparaisons entières
=	X	Egalité entière
<>	X	Différence entière
Affectations et opérations sur les booleens		
:=	X	Affectation booléenne
and	X	Opération AND
or	X	Opération OR
xor	X	Opération XOR
not	X	Opération NOT

Structures de controle		
Structure	Support	Commentaire
if expr then instr	X	Conditionnelle simple
if expr then instr else instr	X	Conditionnelle avec else
while expr do instr	X	Conditionnelle while
		expr est une condition booléenne
		des booléens ou des entiers

Fonctionnalités des tableaux multidimensionnels		
Instruction	Support	Commentaire
array [-x..y] of type	X	Déclaration avec index et intervales
array [-x1..y1, -x2..y2, -x3..y3] of type	X	Déclaration multidimensionnelle
tab[i,j] :=x	X	Affectation à une case d'un tableau
tab[i,j] :=tab[k,l]	X	Affectation à partir d'un tableau
tab[i,j] < tab[i,j]	X	Comparaisons sur les éléments entiers d'un tableau
tab[i,j] or tab[i,j]	X	Opérations booléennes sur les éléments booléens d'un tableau

Fonctionnalités des fonctions		
Instruction	Support	Commentaire
function max (a : int, b : int) : int	X	Déclaration avec argument et valeur de retour
function bsort (ref tab : array(1..10) of int, size : int) : unit	X	Passage d'un tableau en argument via référence (effet de bord)
function bsort (tab : array(1..10) of int, size : int) : unit	X	Passage d'un tableau en argument via copie
Fonctions récursives	X	

Commentaires		
Instruction	Support	Commentaire
(* *)	X	Commentaires simples
(* (* *)	X	Commentaires imbriqués

Fonctions prédéfinies		
Instruction	Support	Commentaire
read var	X	Lecture d'une entrée stockée dans une variable
read tab[i]	X	Lecture d'une entrée stockée dans une case d'un tableau
write var	X	Affichage d'une variable dans la console
write "str"	X	Affichage d'une constantee dans la console
write tab[i]	X	Affichage d'un élément d'un tableau

De plus l'analyse lexicale de notre compilateur se fait sans sensibilité à la casse, ce qui laisse une "marge d'erreur" au programmeur, s'il se trompe et écrit While, wHile ou whiLE...etc ce n'est pas grave notre compilateur saura comprendre son code.

Il est également important de noter que les opérations sur les entiers respectent l'ordre habituel des priorités sans avoir besoin de forcer l'ordre avec des parenthèses.

4 Détails techniques

4.1 Résumé

Voici quelques points techniques que nous avons mis en oeuvre dans ce compilateur :

- Les entiers prennent 4 octets en mémoire, et les booléens que 1.
- Le compilateur ne comporte aucune fuite mémoire.
- Nous avons résolu tous les conflits de la grammaire, elle n'est donc pas ambiguë.
- Lors d'un appel de fonction à l'intérieur même d'une fonction, les variables locales utilisées après l'appel de fonction sont sauvegardées dans la pile afin d'êtres restaurées.

4.2 Tableaux

La difficultées de la mise en place des tableaux était surtout au niveau du calcul de l'index avec les indices.
Par exemple lors d'une affectation *tab(i, j) := 15;* l'index (servant à calculer l'addresse correcte dans le tableau correspondant aux indices i et j) doit être déterminé lors de l'éxecution, en MIPS.

4.3 Fonctions

Les fonctions fonctionnent comme ceci :

1. Lors de l'appel de fonction, les arguments sont empilés dans la pile (*$sp*) à l'envers : le premier argument se trouve en sommet de pile.
2. La fonction stocke ensuite les arguments empilés dans ses propres variables (table des symboles de la fonction) et les dépile.
3. L'adresse de retour (contenue dans *$ra*) est empilée au sommet de pile.
4. Si un appel de fonction a lieu dans cette même fonction, toutes les variables locales utilisées après l'appel de fonction sont empilées dans la pile et sont restaurées ensuite.
5. A la fin de la fonction, le registre *$v0* contient la valeur de retour si la fonction n'est pas de type *UNIT*, et on retourne vers l'adresse de retour sauvegardée et on la dépile.

5 Optimisation

Nous avons implémentés quelques fonctionnalités d'optimisation :

- La suppression des constantes dupliquées afin que le programme soit plus léger, autant sur le disque que dans la mémoire.
- La modification des "arg1 + 0", "arg1 * 1", "arg1 * 0", "arg $\wedge$ 1" en simple affectations afin d'optimiser le temps processeur lors de l'éxecution.

Les optimisations sont par défaut désactivées et peuvent êtres activées avec l'option **–opti**, mais sont automatiquement activées lors des tests (*make text*). Lorsque les optimisations sont activées, quelques informations de debug sont affichées dans la sortie (le nombre de modifications (*change*) par tour (*loop*) d'optimisation).