MiniCC – Projet Compilation

Par Mathis VERMEREN et Nathan SEIGNOLE

MiniCC est un compilateur sommaire de langage C pour une cible MIPS.

Architecture logicielle

Les différents fichiers, correspondant à autant d'étape de compilation, sont :

- **common.c** : décrit **parse_args**, la fonction qui lit les arguments en ligne de commande, ainsi que d'autres petites fonctions utilitaires au programme et au debug.
- **lexico.l** : écrit en lex, ce fichier contient les règles lexicales du compilateur (mots réservés, caractères reconnus comme nom de variable, chaîne, types et opérateurs reconnus) ainsi que la fonction **main** de MiniCC. Cette fonction appelle **parse_args**, **yyparse** pour appliquer les règles lexicales et **analyse_tree**, fonction qui effectue l'analyse syntaxique qui suit.
- **grammar.y** : écrit en yacc, ce fichier contient les règles syntaxiques sous forme d'arborescence, qui permet de transformer les tokens obtenus en analyse lexicale en nœuds utilisables pour la suite. Il contient donc toutes les fonctions de création de nœud, qui, en fonction de la nature du nœud, remplissent les champs nécessaires et allouent les enfants. Ces fonctions sont sous la forme **make_<nature>**. grammar.y contient également la fonction **analyse_tree**, qui appelle le reste de la chaîne de compilation si les flags -s ou -v ne sont pas mis. Cette analyse permet déjà de détecter des erreurs de syntaxe.
- **passe_1.c** : ce fichier décrit **analyse_passe_1**, la fonction principale de la passe 1, l'étape de vérification contextuelle. Cette fonction récursive parcourt l'arbre tout entier, détectant des erreurs de type, de déclaration, ou encore d'ordre d'écriture. Pour les expressions (toute forme de calcul), une fonction spécifique **verif_expr** est donnée : elle lit le tableau **exp_tab** qui associe à un type d'expression un nombre d'opérandes, leur type, et un type de retour, et permet de vérifier que ces conditions sont respectées. Une partie importante de la passe 1 est l'application des contextes, qui permet d'estimer la taille mémoire nécessaire pour un bloc de calcul et permet de détecter les variables en double ou non déclarées. Elle se fait avec une interface donnée dans **miniccutils**.
- **passe_2.c** : ce fichier décrit **gen_code_passe_2**, la fonction principale de la génération de code. A partir d'un arbre de programme correctement réalisé, elle permet de générer du code assembleur dans un objet *program* à partir de l'interface donnée par **miniccutils**. Spécifiquement, cette partie a été réalisée par *reverse engeneering* depuis l'exemple de la partie I du polycopié, figuré par **test.c**, qui donne un bon exemple d'expressions, de boucle, et de print. Pour faciliter l'évaluation d'expression dans un ordre précis (comme par exemple pour les boucles for ou les expressions binaires) et pour utiliser la propriété que tous les enfants d'une expression sont soit une autre expression, un identifiant, ou un littéral,

nous avons réalisé la fonction **gen_code_expr** qui traite spécifiquement les expressions, avec les allocations de registres que cela implique.

Notes sur le script de test

Le script **Tests.py** exécute minicc avec tous les tests qui sortent des erreurs en ligne de commande, c'est-à-dire les tests Syntaxe/KO et Verif/KO.

Notes sur les affichages

Pour une raison qui nous échappe, certains des codes générés par notre minicc n'affichent rien dans la console MARS lors des print – même avec le registre $4 qui pointe sur le bon segment du .data. Nous sommes perplexes.

Notes sur le module d'allocation de registres

L'interface fournie dans dans miniccutils est bien pratique, mais nécessite de l'algorithmique supplémentaire. Il y a plusieurs méthodes de procéder – nous avons choisi une manière plutôt simple qui se base sur le postulat qu'à chaque appel de gen_code_expr, il y a un registre disponible pour écrire le résultat. Cela veut dire que seules les opérations binaires nécessitent d'allouer un registre, pour leur deuxième argument, qui est immédiatement désalloué une fois le résultat obtenu.

La valeur de retour des fonctions **allocate()** et **deallocate()** permet de passer outre la distinction entre registres et temporaires : en effet, côté expression, il suffit juste de savoir dans quel registre mettre le résultat, et la partie sauvegarde/restauration de temporaire ne se présente que sous la forme de lignes de codes avant et après l'expression.

Maintenant, l'implémentation des temporaires est un peu bancale et ne fonctionne pas totalement.

Notes finales

La version de minicc que nous rendons est sommaire mais fonctionnelle. Ce projet a été une bonne occasion de toucher à beaucoup de choses – Mathis est plutôt un développeur haut-niveau qui a l'habitude de travailler sur des problèmes algorithmiques et sur du scripting, tandis que Nathan est plutôt bas niveau. Réaliser MiniCC a donc été un challenge pour tous les deux.