Projet Compilation - Mini Rapport

Clément LEBOULENGER et Julie RAGO

Ce projet a pour but l'écriture d'un compilateur, c'est-à-dire l'écriture d'un programme qui transforme un programme source en programme assembleur.

I - Modules et fonctionnalités

1) Analyse lexicographique :

L'analyse lexicale est la phase de transformation d'une suite de caractères (d'un fichier) en une suite de lexèmes (ou tokens). Dans le cadre de ce projet, nous avons utilisé l'outil Lex afin de réaliser cette transformation.

2) Analyse syntaxique :

L'analyse syntaxique est la phase au cours de laquelle on vérifie que la suite de tokens en sortie de l'analyse lexicale est valide. Pour faire cette analyse, on décrit les langages valides à l'aide de règles de grammaire. L'outil Yacc permet ensuite de générer à partir de la grammaire le code C associé.

C'est au cours de cette analyse que l'on construit l'arbre du programme. L'arbre est ensuite affiché grâce à la fonction **dump_tree()** fournie dans le fichier **common.c**. Cette fonction produit un graphe de l'arbre au format graphviz, qui est visualisable avec **xdot**.

3) Analyse des arguments de la ligne de commande et options

Le programme principal obtenu, appelé **minicc**, est un compilateur MiniC. Certains arguments doivent par conséquent être supportés en ligne de commande. Grâce à l'utilisation de la fonction **getopt()**, nous avons pu implémenter l'usage d'options, afin de par exemple spécifier le nom du fichier assembleur produit, définir le nombre maximum de registres à utiliser, ou bien encore d'arrêter la compilation après une certaine étape.

4) Module de contexte *(annexe)*

Le module de contexte définit le type **context_t** qui réalise l'association entre un nom d'identificateur et la définition de la variable correspondante. Ce module n'a pas été implémenté par nous même et nous a été fourni en annexe, nous avons donc pu utiliser les fonctions de l'interface fournie afin de créer et manipuler un objet de type **context_t**.

5) Module d'environnement *(annexe)*

Le module d'environnement réalise la gestion de l'empilement et du dépilement des contextes. Il permet d'associer un nom de variable à sa définition dans le contexte le plus proche (du bloc le plus interne vers le bloc le plus externe, puis variable globale). Pour cela, le module chaîne entre eux des contextes à l'aide de plusieurs variables statiques en interne.

La difficulté de ce module est le calcul des offsets (en pile ou dans la section .data) des différentes variables du programme. En effet, il faut pour cela pouvoir se souvenir de l'offset correspondant à la fin de l'analyse des variables globales.

6) Allocateur de registres *(annexe)*

Le but de ce module est de fournir les numéros des registres pour les instructions du programme assembleur. La difficulté est de gérer le cas où il n'y a plus de registres disponibles. Dans ce cas, une expression dite temporaire est stockée en pile pour libérer un registre et être restaurée plus tard.

7) Première passe : vérifications contextuelles

L'analyse sémantique est faite au cours de la première passe, soit la première exploration de l'arbre construit. Même si un programme est syntaxiquement correct, il n'est pas forcément correct : en effet un nom de variable peut être utilisé sans avoir été déclaré par exemple. L'ensemble de ces vérifications sont effectuées lors de cette passe. Pour parcourir l'arbre en profondeur, on fait appel à la fonction créée de manière récursive sur les fils de chaque nœud de l'arbre. C'est ici qu'est utilisé le module de contexte, on réalise aussi la vérification des types des variables et expressions grâce à deux fonctions crées, **type_op_unaire()** et **type_op_binaire()**.

8) Deuxième passe : génération de code

La passe de génération de code effectue un parcours de l'arbre au cours duquel sont générées les instructions assembleur du programme. L'allocation des variables locales est gérée par une fonction **block_allocation**, les expressions sont gérées par **expression_handler**, et l'affectation par **affect_handler**. Il en est ainsi pour les *print*, les *for*, *while*, *do while* et les *if*.

II - Fonctionnalités qui ne marchent pas, les bugs connus

Nous avons pu rencontrer des bugs au cours du développement de ce projet, cependant nous n'avons à ce jour aucun bug, du moins à notre connaissance, puisque nous avons réussi à corriger les bugs rencontrés. Il existe très certainement des bugs, mais ceux-ci n'ont pas encore été découverts.

III - Utilisation du script de test

Pour vérifier la bonne exécution de notre compilateur, nous avons réalisé plusieurs séries de tests. Nous avons donc écrit un script de tests en python afin d'automatiser les tests et en vérifier le résultat. Cela nous permet d'avoir des tests de non régression et de s'assurer ainsi qu'un ajout ou une modification dans une passe n'engendre pas de problèmes dans une fonctionnalité qui marchait.

Pour lancer le script, il suffit de compiler dans un premier temps notre **minicc** grâce au Makefile fourni dans le dossier **src**, puis d'exécuter la commande suivante : **python tests.py**

Ainsi, le résultat d'exécution de chaque test pour chaque étape s'affichera.

Attention : Pour la génération de code, on compare également le résultat de l'exécution sur Mars de notre fichier assembleur produit avec celle du compilateur de référence fourni. Il sera donc nécessaire d'ajouter au dossier le binaire **minicc_ref**.