Projet de Compilation
Licence d'informatique
–2020 - 2021–

Table des matières

Johnson VILAYVANH
Groupe 2

LE PROJET

DESCRIPTION DU PROJET

Le projet est un compilateur utilisant les outils « flex » pour l'analyse lexicale, et « bison » pour l'analyse syntaxique et la construction de l'arbre abstrait.
L'analyseur est programmé pour fonctionner avec des fichiers de type « TPC » qui est un sous-ensemble du langage C.
Le langage cible est l'assembleur « nasm » dans sa syntaxe pour Unix (option de compilation -f elf64) avec les conventions d'appel AMD 64.
Pour comprendre comment est structuré un programme TPC et sa grammaire, se référer au pdf du projet joint dans le dossier du projet.

MANUEL UTILISATEUR

Organisation

Le répertoire est composé de 5 sous-répertoires :

- src pour les fichiers sources écrits par les humains,
- doc pour le rapport,
- bin pour le fichier binaire (votre compilateur doit être nommé tpcc),
- obj pour les fichiers intermédiaires entre les sources et le binaire,
- test pour les jeux d'essais, avec des sous-répertoires :

Lancement du compilateur

Afin de lancer le programme, il suffit d'utiliser le makefile joint dans le dossier en lançant la commande « make » qui permet de créer l'exécutable « tpcc » qui sera situé dans le fichier « bin ».

Le compilateur utilise des lignes de commande :

- Ligne de commande : ./bin/tpcc [OPTIONS] < fichier.tpc (pour récupérer en entrée standard les données du fichier.tpc et les renvoyer vers le programme).

Les [OPTIONS] étant :

-t, --tree affiche l'arbre abstrait sur la sortie standard
-s, --symtabs affiche toutes les tables des symboles sur la sortie standard
-h, --help affiche une description de l'interface utilisateur et termine l'exécution

Le code cible est généré dans le fichier « _anonymous.asm », et peut donc être exécuté de la manière suivante :

```
nasm -f elf64 _anonymous.asm
gcc -o asm _anonymous.o -no-pie
./asm
```

Les jeux de tests

Comme expliqué précédemment, le dossier du projet présente un répertoire nommé « test » dans lequel on va trouver les sous-répertoires :

- good : tests qui ne produisent aucun message d'erreur (valeur de retour = 0)
- warn : tests qui produisent un warning mais crée l'exécutable (valeur de retour = 0)
- syn-err : tests qui produisent une erreur sémantique (valeur de retour = 2)
- sem-err : les autres erreurs : ligne de commande, mémoire insuffisante, fonctionnalité non implémenté (valeur de retour = 3)

Un script bash « test.sh » dans le répertoire « src » du projet permet de tester l'intégralité des fichiers TPC présent dans les sous-répertoires du répertoire « test ». Le script va également lancer la compilation du makefile avant de tester chacun des fichiers, au cas où l'exécutable « tpcc » n'avait pas été compilé.

Pour lancer le script bash, il suffit d'entrer la ligne de code « bash ./src/test.sh »

PARTIE DÉVELOPPEUR : DIFFICULTÉS ET CHOIX

LES DIFFICULTÉS RENCONTRÉES

Les structures

L'utilisation des structures m'a posé un réel problème durant ce projet.
Dans un premier, il fallait à chaque déclaration d'une variable de type structure (et non la déclaration de la structure en elle-même) déclaré l'intégralité de ses champs.
Prenons par exemple la structure suivante :

```
struct point {
        int x;
        int y;
};
```

Si on déclare une variable de cette structure, par exemple la variable « nomVar », il fallait, si la variable était globale, déclarer dans la section .bss tous les champs de cette variable :

```
nomVar.x : resd 1
nomVar.x : resd 1
```

Ou bien si cette variable est locale, réserver l'espace nécessaire dans la pile.

Ensuite, il fallait également que l'utilisation des champs de la variable soient reconnus par l'analyseur lexicale. En effet, les variables s'utilisent en écrivant « nomVar.x », « nomVar.y ». Or mon analyseur lexicale ne reconnaissait pas des identificateurs avec un « . » au milieu de celui-ci.

Enfin, lorsqu'on veut qu'une variable de structure soit intégralement associée à une autre (« var1 = var2 ; », avec var1 et var2 des structures), il fallait copier l'intégralité des champs de var2 dans les champs de var2.

L'appel à la fonction scanf

La fonction « scanf » est utilisée pour les fonctions « reade » et « readc » qui permettent de lire un nombre ou caractère au clavier. Lors d'un appel à scanf (call scanf), il est nécessaire d'aligner la pile pour avoir un multiple de 16. L'alignement exact de la pile m'a posé problème car dans le cas où on a alloué une pile d'une taille par exemple impair (sub rsp, 7), il faut à l'appel de scanf, avoir une pile d'un multiple de 16, et le code nécessaire pour avoir cet alignement de pile m'a posé problème.
De plus, il fallait également l'utilisation de registres avec leurs adresses : or si on veut reade ou readc une variable locale, étant donné que cette variable locale se trouve sur la pile, je ne savais pas comment charger l'adresse d'une variable sur la pile dans le registre « rsi » pour ainsi associer la valeur entrer au clavier à l'adresse de cette variable.

Return d'une fonction qui ne retourne pas « void »

Lorsqu'une fonction ne retourne pas « void », il fallait faire très attention à ce qu'on ait la présence d'un return dans la fonction. Cependant, il ne fallait pas simplement vérifier la présence d'un « return » n'importe où dans la fonction car par exemple, si ce « return » se trouve dans un « if » inatteignable, alors on ne va jamais return de valeur, alors qu'on a pourtant bien un « return » dans la fonction.
Il fallait donc faire la distinction entre les « return » qui peuvent être inatteignable (donc par exemple dans un « if »).

CHOIX POUR RÉSOUDRE LES DIFFICULTÉS

Structures non résolues

Malheureusement l'intégralité des difficultés rencontrés avec les structures n'a pas été résolue par manque de temps.

Résolution de l'appel à scanf

Pour aligner la pile, j'ai choisi une solution qui n'est pas réellement optimisée. En effet, lorsqu'on étudie l'intégralité des variables locales au début de chaque fonction, on rajoute à la valeur X « sub rsp, X » avec laquelle on alloue la pile assez pour que X soit un multiple de 16.
Ainsi, si on devait allouer X = 7, on ajoute 9 pour que 7 + 9 = 16, et donc qu'on puisse allouer un multiple de 16. Les 9 derniers bytes alloués ne seront ainsi pas réellement utilisé.
J'utilise aussi les variables pré-établies « scanfInt » et « scanfChar » qui permettent de stocker leur adresse lorsqu'on veut lire respectivement un int ou char, et on met leur résultat dans la variable d'origine dans le read(e/c).

Présence de variables globales

Dans mon programme, on retrouve une énorme quantité de variable globale. En effet, pour retenir les différentes informations entre chaque fonction qui traverse chaque nœud de la pile, on utilise ces variables globales.
Ainsi, on a par exemple une variable « g_first_variable », qui indique qu'on a récupérer la 1ère variable sur la ligne (au cas où on en a de nouveau besoin si on fait un Assignement (étant donné qu'on parcourt ici l'arbre en parcours suffixe, on récupère la 1ère variable de la ligne en 1er et pas en dernier)). Ou bien la variable « g_last_registre » et « g_last_param_registre » qui permettent de savoir quels registres ont déjà été utilisé et donc quelle est la suivante qui doit être utilisé. Également les tableaux « local_registre_4bit », « local_registre_1bit », «param_registre_4bit », «param_registre_1bit » qui stockent l'intégralité des registres nasm.
J'utilise également les variables globales pour résoudre le problème de return d'une fonction qui ne retourne pas « void » : dès lors qu'on rentre dans un « if » ou « while », j'utilise les variables « g_in_if » ou « g_in_while » pour savoir si on est dans un « if » ou « while ». Ainsi, si on y retrouve un « return », on fait en sorte que la variable « g_returned » soit toujours égale à 0 (variable qui nous dit si on a déjà rencontré un « return » dans la fonction).

De cette façon, j'utilise donc ces nombreuses variables globales pour le bon fonctionnement de la création du fichier cible en nasm.