Université Gustave Eiffel

BERNIER Aurélien
BLONDEL Lucas

Rapport Projet compilation

Sommaire

I. Introduction

Présentation

L'objectif du projet était de réaliser un compilateur d'un langage donné dérivé du C à partir du projet d'analyse syntaxique du semestre précédent. Le but était aussi de gérer les différentes erreurs sémantiques en construisant les tables de symboles et enfin traduire le langage tpc en nasm.

Avancement du projet

Nous avons réalisé quasiment toutes les parties du projet demandées sauf la traduction des fonctions à plus de 6 paramètres ou encore le passage des tableaux en paramètre (plus de détails dans la partie Difficultés), nous avons également obtenu un score de 93,11 / 100 au bac à sable avec 13 tests d'erreurs sémantiques qui ne passent pas. Les options demandés sont aussi implantés : -t, -h –help, –tree, -s et --symtabs.

Répartition des tâches

Lucas s'est occupé de de la plupart des erreurs sémantiques et de la construction des tables de symboles tandis qu'Aurélien a fait en grande partie la traduction en nasm. Nous avons fait le jeu de test à deux.

II. Mise en œuvre

Généralités

La compilation d'un programme se déroule en plusieurs étapes :

- Un 1er parcours de l'arbre abstrait qui va permettre de construire les tables de symboles et signaler les erreurs sémantiques de redéfinition (variable déjà déclarée ou fonction redéfinie). Ici, on utilise le module table.h.

- Un 2ème parcours permet de vérifier les autres erreurs sémantiques comme celles de type, main non déclaré ou variables non déclarées. Ici, on utilise le module sem.h.

- Un dernier parcours va s'occuper de la traduction de l'arbre en programme nasm. Ici, on utilise le module trad.h.

Ce fonctionnement nous permet de faire la traduction de l'arbre en partant du principe que le programme tpc est correct en terme de sémantique et de syntaxe tout en divisant les tâches par thème.

Tables de symboles

La construction des tables de symboles se fait en ne parcourant que le nœud déclaration des fonctions (et du programme pour les variables globales).

Pour simplifier, nous avons décidé par convention que la table 0 serait celle des variables globales et que les tables suivantes seraient celles des fonctions. De plus si une fonction ne contient ni paramètre ni variable locale, la table est tout de même construite mais est laissée vide.

Les informations concernant une fonction comme son type de retour ou nom sont stockées dans la table des variables globales.

Les fonctions getchar, getint, putchar et putint ne doivent pas être redéfinies et par conséquent, nous devons parcourir exceptionnellement les nœuds instructions des fonctions et si une de ses fonctions est utilisée une table correspondante est automatiquement construite.

Exemple de table :

```
int tab[5], a;

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int b;
    b = 4;
    a = 2;
    putint(getint());
    return sum(a, b);
}
```

Figure 1: Programme écrit en tpc

```
Table 0 :
|Type : int array, Ident : tab, Taille : 5, Fonct : non, Adr : 0
|Type : int, Ident : a, Taille : 0, Fonct : non, Adr : 40
|Type : int, Ident : sum, Taille : 0, Fonct : oui, Adr : 48
|Type : int, Ident : main, Taille : 0, Fonct : oui, Adr : 56

Table 1 :
|Type : int, Ident : a, Taille : 0, Param : oui, Adr : -8
|Type : int, Ident : b, Taille : 0, Param : oui, Adr : -16

Table 2 :
|Type : int, Ident : b, Taille : 0, Param : non, Adr : -8
```

Figure 2: Tables produites à partir du programme TPC figure 1 (sans appel à getint et putint)

```
Table 0 :
|Type : int array, Ident : tab, Taille : 5, Fonct : non, Adr : 0
|Type : int, Ident : a, Taille : 0, Fonct : non, Adr : 40
|Type : void, Ident : putint, Taille : 0, Fonct : oui, Adr : 48
|Type : int, Ident : getint, Taille : 0, Fonct : oui, Adr : 56
|Type : int, Ident : sum, Taille : 0, Fonct : oui, Adr : 64
|Type : int, Ident : main, Taille : 0, Fonct : oui, Adr : 72

Table 1 :
|Type : int, Ident : n, Taille : 0, Param : oui, Adr : -8

Table 2 :
VIDE

Table 3 :
|Type : int, Ident : a, Taille : 0, Param : oui, Adr : -8
|Type : int, Ident : b, Taille : 0, Param : oui, Adr : -16

Table 4 :
|Type : int, Ident : b, Taille : 0, Param : non, Adr : -8
```

Figure 3: Tables produites à partir du programme TPC figure 1 (avec appel à getint et putint)

Nous pouvons noter plusieurs éléments :

- le type d'une variable ou fonction et représenté par un entier de 0 à 4. 0 correspond à void, 1 à char, 2 à int, 3 à char array et 4 à int array. Nous avons ajouté 2 types différents pour les tableau pour simplifier la vérification de type plus tard.

- une table de fonction sans paramètre ni variable locale est simplement noté VIDE (ici getint)

- le champ Taille correspond à la taille d'un tableau, il va donc valoir 0 dans la cas d'une variable

- dans les tables de fonction, le champ param indique si la variable est un paramètre. De même dans la table globale le champ fonct indique si il s'agit d'une fonction.

- adr correspond à l'adresse relative de la variable dans la table de la fonction. Dans la table globale il s'agit d'une adresse globale. Afin de simplifier la traduction en nasm, les adresses des variables dans les tables de fonctions sont négatives car les adresses de variables locales sont négatives par rapport à rbp.

- par convention, les fonctions sont toujours après les variables locales dans la table globale et les paramètres sont toujours avant les variables locales dans les tables des fonctions.

- les tables des fonctions sont dans le même ordre que les fonctions dans la tables globale afin de pouvoir déterminer facilement l'indice d'une table à partir du nom de la fonction ce qui est utile pour la traduction.

Erreurs sémantiques

Après la construction des tables, nous pouvons passer aux erreurs sémantiques qui sont gérées par les fonctions de sem.c. Ces fonctions sont appelées plus tard dans le module trad.c dans la fonction traduction.

Dans ce module, nous vérifions les types de chaque expression utilisée et leurs composantes (variables, paramètres ou fonctions) par rapport au cas d'utilisation (affectation, type de retour ou type de paramètres). De là nous émettons un warning, une erreur de sémantique ou rien selon le retour des fonctions.

En plus des types nous vérifions si les fonctions ou variables utilisées sont déclarées, si les tableaux ne sont pas utilisés sans crochet ou si le main est correctement déclaré.

A l'origine, ces fonctions et les fonctions de traductions étaient dans le même module (trad.c) mais nous avons décidé de séparer les deux selon leur tâche pour alléger trad.c et rendre le code plus lisible.

Traduction en nasm

La dernière étape de compilation, et de loin la plus compliquée selon nous, est la traduction du code en nasm.

Comme dit précédemment, avoir séparer vérification d'erreur sémantique et traduction nous permet de partir du principe dans ce module que le code tpc est correct syntaxiquement et sémantiquement sans devoir alourdir les fonctions de traductions avec des vérifications.

Nous avons également décidé de simplifier la taille des types en partant du principe que les entiers et caractères seraient sur 8 octets et non 4 et 1. Cette simplification nous permet de n'utiliser que des registres de taille 8 octets pour n'importe quelle variable. Il faut tout de même noter que cela n'affecte pas les erreurs sémantiques et warning possibles (int dans char etc ..).

Pour simplifier aussi l'utilisation des paramètres dans une fonction, nous les traitons comme des variables locales et stockons leurs valeurs à une adresse relative à rbp.

La traduction débute dans la fonction traduction() où les erreurs sémantiques sont vérifiées. Si aucune erreur n'est détectée, la traduction en nasm commence en traduisant fonction par fonction avec un cas spécial pour le main qui est traduit sous l'étiquette _start. Une fonction de recherche de traduction va en suite appeler la fonction de traduction adéquate selon le nœud rencontré (if, while, affectation, appel de fonction …).

Les fonctions getint, putint, getchar et putchar devant être écrite automatiquement par le compilateur, nous avons 4 fonctions qui écrivent les fonctions directement dans le fichier nasm selon si elles sont utilisées ou non dans le programme tpc d'origine.

Enfin, nous rappelons que nous avons réussit à traduire quasiment tout type de programme tpc sauf ceux comportant des fonctions à plus de 6 paramètres (ces derniers serton juste ignorés et non initilisés) et les fonctions avec un tableau en tant que paramètre (comme int lenTab(int tab[] par exemple).

Jeu de tests

Nous avons repris les tests du projet d'analyse auxquels nous avons ajouté quelques tests good comme par exemple des tests avec fonctions récursives (fibo / factorielle) et des fonctions indirectement récursives. Nous avons aussi corrigé les tests good qui n'étaient plus acceptés notamment à cause de l'absence d'un main.

En plus de ces ajustements et comme demandé dans le sujet, nous avons ajouté une trentaine de tests d'erreur sémantique et 5 tests warning.

Enfin, nous avons mis à jour le script test.bash pour déployer les tests et rédiger un rapport contenant pour chaque fichier testé, le nom et le code de retour avec à la fin un score total.

Au final, le jeu de test comporte 92 tests et nous pensons avoir couvert les points d'erreur important.

Corrections apportées au projet d'analyse syntaxique

Pendant la réalisation du projet nous avons remarqué plusieurs problèmes provenant du lexer et de la construction de l'arbre.

D'abord dans le lexer, nous avions mal défini le paterne de supérieur ou égal. A l'origine nous avions « => » au lieu de « > = » ce que nous avons modifié.

Également, lorsqu'un caractère spécial comme \n était reconnu nous ne donnions à bison que le \ au lieu du caractère spécial.

Nous avons aussi déplacé le main du programme qui se trouvait dans le fichier bison, dans un fichier à part.

Enfin, lors de la construction de l'arbre quand une fonction était utilisée sans paramètre, le code associé à la règle correspondante était vide ce qui provoquait une erreur de segmentation car le nœud de l'arbre n'était pas vide. Nous avons donc ajouté $$=NULL ; à la règle bison et une vérification de nœud NULL.

III. Conclusion

Difficultés et bugs connus

Bien que la partie de construction des tables et la partie de détection d'erreur ne nous ont pas posé de gros problème, nous avons eu de plus grosses difficultés sur la partie de traduction.

Tout d'abord les if et while nous ont posés quelques problèmes notamment d'étiquette. Pour palier à se problème nous avons ajouté des variables globales pour compter et identifier les if, while et expressions conditionnelles (nb_if, nb_while et nb_cond).

Les choses se sont compliquées en arrivant sur la traduction des expressions conditionnelles complexes. En effet, nos appels récursifs n'étaient pas les bon et les affectations de valeur n'étaient pas correctes, ce qui causait des retours faux du programme nasm.

Un autre problème a été avec les variables locales et notamment la recherche de leurs adresses. En effet, l'ajout des fonctions d'entrées sorties dans les tables de symboles avaient créés des décalages et comme nous nous servons des indices des tables en fonction de la position des fonctions dans la table globale, le programme renvoyait la mauvaise adresse pour les variables locales à cause du décalage.

Enfin la dernière difficulté a été la traduction des fonctions et de leurs paramètres. A l'origine nous n'empilions pas les paramètres dans le bon ordre causant des inversions de valeurs. Également, ils n'étaient pas traités comme des variables locales et étaient conservés dans les registres (rdi, rsi, rdx …) ce qui était problématique si une autre fonction était appelée dans une fonction et que les paramètres étaient utilisés après cet appel. Les valeurs étant stockées et utilisées dans les registres elles étaient modifiées après l'appel ce qui n'est pas un comportement attendu. Pour palier à ce problème nous avons simplement utilisé les paramètres comme des variables locales en ajoutant du code nasm qui stocke les valeurs contenues dans les registres à une adresse relative à rbp.

Apports du projet

Malgré les difficultés rencontrées, le projet est tout de même resté intéressant sans être trop compliqué et il nous a permis de comprendre ce qu'il se passe lors de la compilation d'un programme . Le fait d'avoir un suivit en TP nous a permis d'avoir des heures dédiées au projet tout en appliquant directement ce qui a été vu en cours au projet.

Annexe : Compilation et utilisation du compilateur

Tout comme le projet d'analyse syntaxique, il suffit simplement d'utiliser la commande : make

Pour utiliser le compilateur, il faut utiliser : ./bin/tpcc [*OPTION*] < *[FICHIER*]
Les options sont comme au projet précédent -t / --tree pour afficher l'arbre syntaxique et -h / --help qui affiche un court mode d'emploi du programme en plus de l'option -s/--symtabs qui affiche les tables de symboles. Tout paramètre non reconnu affichera le bon usage du programme.

Pour lancer le jeu de tests, il suffit d'utiliser la commande : bash test/test.bash.