Université Gustave Eiffel

Le 28/05/2022
L3 Informatique

Semestre 6

Rapport Projet COMPILATION

Binôme:

RAMAROSON RAKOTOMIHAMINA Johan
MENAA Mathis

Encadrants:

Le corps d'enseignement
de Compilation

Introduction **3**
Contexte 3
Implémentation 3
Manuel d'utilisation **4**
Exemples / Jeu de test **5**
Jeu de test 5
Pierre-Feuille-Ciseau 5
Difficultés rencontrées **6**
Traduction vers un Fichier NASM 6
Optimisation de la gestion des erreurs sémantiques 6
Warning : Reaches end of non-void function 7
Erreurs sémantiques et Warnings traités **8**
Améliorations possibles **10**
Optimiser le parcours de l'arbre 10
Warning : Unused parameter in the function x 10
Stockage des valeurs dans la pile 10
Conclusion **10**

Introduction

En se basant sur le projet d'analyse syntaxique de 2021-2022, l'objectif était d'écrire un compilateur en langage C avec comme langage source le **TPC**.
Le langage cible choisi étant l'assembleur **NASM** dans sa syntaxe pour **UNIX**, le résultat de la compilation sera vérifié en exécutant le code obtenu.

Contexte

TPC signifie 'très petit C', et comme son nom l'indique un sous-langage du langage déjà-existant : le C.
Nous y retrouverons la même syntaxe et les mêmes règles sémantiques de bases du langage père. *Tpcc* est le compilateur que nous devons développer afin de pouvoir compiler et créer un exécutable pour une source **TPC**.

Implémentation

L'implémentation va donc se dérouler en trois grandes phases:

1. *Définition de la syntaxe global du langage, nous utiliserons pour structure de donnée un arbre fils/frère afin de pouvoir parcourir le contenu du fichier suivant la syntaxe prédéfinie.*

2. *Parcours de l'arbre, permettant de définir les règles sémantiques du langage. Ces règles sont, à quelques exceptions près, les mêmes que celles du langage C.*

3. *Traduction du langage, nous parcourons l'arbre une nouvelle fois afin de générer le bytecode du langage.*

Manuel d’utilisation

Avant de pouvoir utiliser le langage, assurez-vous d’avoir gcc d’installé sur votre machine.

Vous pouvez simplement exécuter la commande **“make”** à la source du projet pour générer le compilateur situé dans le répertoire *“bin”*.

Ainsi, depuis la racine pour compiler votre fichier.tpc vous pouvez exécuter la commande suivante :

```
./bin/tpcc < “votreSource.tpc”
```

Le compilateur lit de base l’entrée standard, il est également possible de spécifier le fichier source en argument au compilateur comme ceci :

```
./bin/tpcc <votreSource.tpc>
```

Des options supplémentaires sont également disponibles :

-t/–tree	Permet d’afficher sur la sortie standard l’arbre syntaxique du fichier cible. Attention, l’arbre ne s’affiche que si le fichier est syntaxiquement correct. Sinon une erreur sera détectée avant et arrêtera le programme.
-s/–symtabs	Permet d’afficher la table des symboles sur la sortie standard.
-h/–help	Permet d’afficher la liste des options disponible du compilateur.
-n/-noexec	Permet d’empêcher la génération du bytecode exécutable, essentiel pour les tests notamment.

Après compilation, si votre fichier ne comporte aucune erreur syntaxique ou sémantique, il vous sera alors automatiquement généré un fichier binaire permettant de lancer votre programme.

Par défaut le fichier cible s'intitule **“out”** et est disponible depuis la racine du projet.

```
./out
```

Exemples / Jeu de test

Jeu de test

Un banc de tests est disponible dans le répertoire *test.* Ils sont rangés dans les répertoires suivants :

- **good :** Tous les fichiers de ce répertoire sont valides et ne renvoient aucune erreur sémantique ni aucun warnings
- **syn-err :** Tous les fichiers de ce répertoire renvoient une erreur syntaxique ou lexicale et renvoient 2.
- **sem-err :** Tous les fichiers de ce répertoire renvoient une erreur sémantique et ne lancent donc pas l'exécutable.
- **warn :** Tous les fichiers de ce répertoire renvoient au moins un avertissement sémantique mais ne produisent pas d'erreur et permettent donc de générer un fichier exécutable.

Pour lancer l'ensemble des tests il suffit de lancer le script bash ***test.sh*** qui vous permettra de pouvoir lancer l'ensemble des tests et d'avoir un feedback complet sur la sortie standard.

```
bash test.sh
```

Ce script générera un fichier "feedback.txt" qui permettra d'avoir un compte rendu sur le nombre de fichiers qui sera passé correctement sur le nombre de fichiers total du répertoire. Il testera entre autres les 4 répertoire en fonction de la valeur de retour qu'elle est censée renvoyer.

Pierre-Feuille-Ciseau

Pour tester notre compilateur, un fichier d'exemple se situant dans le répertoire *example/pfc_game.tpc* est donnée, vous pouvez le lancer avec la commande :

```
./bin/tpcc < ./example/pfc_game.tpc
```

Après avoir généré le compilateur tpcc via la commande make comme indiqué ci-dessus.

La commande vous conduira vers une réplique du jeu *"pierre-feuille-ciseau"* contre l'ordinateur en ayant au préalable choisi une graine afin de générer une suite aléatoire de celui-ci.

A chaque round il vous suffira de choisir un coup à jouer et le jeu s'arrête au bout de 5 round. Le gagnant sera indiqué à la fin.

L'exemple n'est pas un bon exemple de code source "propre" du langage mais est surtout présent pour tester un maximum de cas ou d'instructions du langage.

Difficultés rencontrées

Traduction vers un Fichier NASM

La principale difficulté du projet a été la traduction en nasm d'un fichier syntaxiquement et sémantiquement correct.
Et particulièrement la traduction des "et" et "ou" logiques permettant d'effectuer les conditions des instructions "if" et "while".

Notre problème principal était de prendre en main le langage nasm, en prenant une feuille et un stylo nous nous sommes vite rendu compte qu'il nous fallait des variables globales qui s'incrémente à chaque passage d'une instruction logique pour nommer et à jump dans les bon labels lorsqu'il le fallait.

Optimisation de la gestion des erreurs sémantiques

Une autre difficulté a été de parcourir toutes les erreurs sémantiques du langage en minimisant le parcours de l'arbre.
Par exemple, pour une détection de warning de type **"unused / unintialized variable"** la logique pour renvoyer le bon warning au bon moment a été difficile à mettre en place.

Pour ce faire nous avons simplement accordé un **"état"** pour chaque variable à parcourir.
Ainsi chaque variable déclarée possède un état boolean **"Initialisée"** et **"utilisée"**.

Pour chaque variable nous parcourons la fonction où elle est déclarée et nous changeons l'état de la variable en fonction de ce que nous rencontrons dans le parcours.
Le bon warning est renvoyé en fonction des états renvoyés par le parcours.

Une amélioration possible de cette méthode et moins coûteuse (en moins de parcours) aurait été d'utiliser la table des symboles de la fonction, d'attribuer les états dans un champ supplémentaire de la structure de symbole et d'ainsi faire les tests en récupérant la variable en temps constant. Cette méthode permet ainsi de ne parcourir qu'une seule fois l'arbre pour détecter toutes les erreurs de toutes les variables.

Warning : Reaches end of non-void function

Un autre warning nous a été difficile à mettre en place : **“reaches end of non-void function”.**
Notre principal problème ici était de cibler tous les cas où une fonction renvoie cette erreur, pour ce faire il nous a fallu prendre le problème localement.
Pour chaque ensemble d’instruction il fallait vérifier que cet ensemble permettait à la fonction d’avoir un état de retour cohérent.

Le problème se résume avec l’approche suivante :

1) Une instruction **if** simple n’a aucune incidence sur l’état de retour de la fonction car même si elle possède un **return** et que la fonction n’en possède pas, l’état de retour de la fonction n’est pas cohérent.

2) Même raisonnement pour l’instruction **while**, si un **return** est présent dans l’instruction, celà n’a pas d’incidence sur l’état de retour de la fonction.

3) Une fonction est cohérente si elle possède un **return** dans son bloc d’instruction principal.

Sinon, en passant les approches ci-dessus, nous pouvons conclure que les seuls cas, dans le niveau de notre compilateur, où la fonction est susceptible d’envoyer ce warning sont les cas suivants :

- Lorsque la fonction possède dans son bloc principal, un **if** suivi d’un **else**. Car si tel est le cas, le programme suppose que l’on peut rentrer dans le **if** ou dans le **else**. Par conséquent les deux instructions doivent avoir un état de retour cohérent. Le programme est donc dans un état cohérent si le **if** et le **else** sont dans un état cohérent.

- Lorsque la fonction possède dans son instruction principal un **“switch”**, pour que la fonction soit cohérente il faut que tout les cases et le default soient cohérent s’ils existent.

Ces deux approches sont toutes deux récursives tant que l’on rencontre ces cas dans les instructions à tester.

Il suffit ensuite d'effectuer un parcours dans l’arbre afin de traiter ces cas suivant l’instruction que nous rencontrons.

L'algorithme se traduit ensuite comme suit :

1. Au début de chaque fonction si la fonction renvoie quelque chose, on peut s'arrêter là et on renvoi true.
2. Sinon on parcourt les instructions dans l'arbre, si on tombe sur un **if** ET que ce **if** à un **else**, on regarde si ce **if** à un état cohérent (i.e on relance l'algorithme sur ce **if** ET sur ce **else**), si ce n'est pas le cas on **return** false.
3. Si on tombe sur un **switch** on vérifie si tout les cases ET son default si il y en a un sont dans un état de **return** cohérent.
4. Si l'algorithme renvoie false pour la fonction, on envoi l'avertissement.

Erreurs sémantiques et Warnings traités

Voici les **erreurs sémantiques** que notre compilateur traite (erreur renvoyant 2) :

Nom de l'erreur	Description
Undefined reference to variable 'X'	Détecte si une variable a bien été déclarée avant d'être utilisée
Invalid symbol id : 'X', already referenced as global	Détecte une fonction n'a pas le même nom qu'une variable global
Can't find reference to 'main'	Cas où il n'y a pas de fonction main dans le programme
Expected main to return an integer	Cas où la fonction main ne retourne pas un int
Not enough arguments given to function 'X'	Cas où l'on ne donne pas le nombre d'arguments attendu à un appel de fonction.
Too much parameters while trying to call function 'X'	Cas où l'on donne trop de paramètres à l'appel de la fonction X
Attempt to call a void-function as parameter	Apparaît lorsque l'on tente d'appeler en argument d'une fonction une fonction retournant void
Undefined reference to function : 'X'	Apparaît lorsque l'on tente d'appeler une fonction qui n'existe pas
Duplicate case value	Apparaît lorsqu'un switch possède 2 valeurs de cases équivalente

Multiple default label in 1 switch	Apparaît lorsque l'on essaye d'attribuer plusieurs default à un seul switch
Void function not ignored as it ought to be	Lorsqu'une fonction return une fonction de type void

Voici ensuite la liste des **Warnings** que nous avons traités, pour rappel un warning n'empêche pas l'exécution du programme.

Nom du warning	Description
Assigning variable 'X' to an integer	Apparaît lorsqu'on affecte un type int (variable ou constante) à une Lvalue de type char
Variable 'X' set but no used	La variable X est déclarée, elle est affectée à une RValue mais n'est pas utilisée dans le programme
Unused variable 'X'	La variable X est déclarée mais n'est ni affectée ni utilisé dans la fonction
'X' is used uninitialized in this function near line 'N'	La variable X a été déclarée, et est utilisée dans la fonction sans affectation. A noter que l'avertissement est effectif uniquement si au moment de l'utilisation la variable n'est pas affectée. A partir du moment où la variable concernée est déclarée le warning ne peut plus survenir dans la fonction pour cette variable.
Control reaches end of non-void function	Une fonction qui return un type tpc int ou char possède un état incohérent, c'est à dire que la suite d'instruction de la fonction ne permet pas de renvoyer systématiquement une valeur.
Return with a value, in function returning void	Une fonction retourne un entier alors que sa valeur de retour est de type char.
Return with no value, in non-void return function	Une fonction possède un return sans valeur alors qu'elle est censée en retourner une.

Améliorations possibles

De nombreux axes d'améliorations sont possibles dans notre projet, en voici quelques exemples des plus flagrants.

Optimiser le parcours de l'arbre

En premier lieu nous aurions pu prévoir un meilleure système de parcours d'arbre, de meilleurs algorithmes afin d'éviter le plus possible d'en faire et ainsi d'optimiser au mieux la compilation.

Warning : Unused parameter in the function x

Parmis les nombres warning à détecter par le compilateur certains nous ont donné plus ou moins de difficultés. Plus on ajoutait des exceptions, plus il fallait faire attention à ne pas avoir de conflit entre les erreurs/warnings déjà implémentés.
Par exemple l'erreur sémantique qui génère un warning dans le cas où un paramètre d'une fonction n'a pas été utilisé dans celle-ci n'a pas été implémenté.
Nous aurions donc très bien pu implémenter le warning "unused parameter in the function x".

Stockage des valeurs dans la pile

Une optimisation très intéressante à effectuer aurait été de travailler avec des registres de la même taille que le type des données que l'on manipule. Dans le projet, nous avons décidé de travailler avec des registres de taille 8 octets et ceux peu import la taille du de la variable utilisée. Nous aurions pu manipuler les types avec leur vrai taille par rapport au langage C, à savoir : 4 octets pour les entiers et 1 octet pour les char.

Conclusion

Dans son ensemble, le projet nous à permis de mettre en pratique toutes les connaissances vu en compilation ce semestre.

En passant par l'utilisation d'un langage très bas niveau : l'assembleur jusqu'à la compréhension au sens propre et fonctionnel d'un Compilateur.

Le projet a été très agréable à réaliser et surtout changeant des projets habituels que l'on a pu concevoir depuis notre entrée en licence.
Ici nous créons notre propre compilateur, même si la syntaxe a été réalisée par les professeurs, la gestion des règles sémantiques, la construction de l'arbre et la manière de générer le code assembleur relève de nos travaux.

Malgré tout cela, la partie la plus frustrante reste la gestion de la pile en NASM. Nous avons trouvé que ce concept presque central dans la conception d'un programme NASM à été peut être pas assez approfondi en cours.

Le projet nous a permis de mieux comprendre comment elle fonctionnait mais c'est sur cette partie que nous avons passé le plus de temps.