Projet de Compilation

Yvân TRAN, KHELIFI Allan

Table des matières

Chapitre 1

Introduction

Le projet que nous avons réalisé en binome est l'écriture d'un compilateur d'un langage source **TPC** (sous-ensemble du langage C) vers un langage cible **NASM 64** à l'aide des outils **flex** et **bison**.

Le but de ce compilateur étant de traduire un programme TPC fourni, si celui-ci respecte la grammaire du langage. Le fichier ASM généré pourra être ensuite compilé pour en faire un exécutable. Si le programme fournit ne respecte pas la grammaire du langage **TPC**, les messages d'erreurs et/ou avertissements seront affichés en conséquence.

1.1 Le langage TPC

Notre compilateur se voit être capable de prendre en entrée un programme dans le langage **TPC** et le traduire en NASM 64. Le langage **TPC** étant un sous-ensemble du langage C autorisant l'écriture de programme avec des fonctions et des appels de fonctions, des variables et pointeurs, des fonctions implémentées par le langage permettant de lire la saisie utilisateur dans une variable ainsi qu'afficher le contenu d'une variable.

```
Prog          : DeclVars DeclFoncts ;
DeclVars      : DeclVars TYPE Declarateurs ';'
              | ;
Declarateurs  : Declarateurs ',' IDENT
              | Declarateurs ',' '*' IDENT
              | IDENT
              | '*' IDENT ;
DeclFoncts    : DeclFoncts DeclFonct
              | DeclFonct ;
DeclFonct     : EnTeteFonct Corps ;
EnTeteFonct   : TYPE IDENT '(' Parametres ')'
              | TYPE '*' IDENT '(' Parametres ')'
              | VOID IDENT '(' Parametres ')' ;
Parametres    : VOID
              | ListTypVar ;
ListTypVar    : ListTypVar ',' TYPE IDENT
              | ListTypVar ',' TYPE '*' IDENT
              | TYPE IDENT
              | TYPE '*' IDENT ;
Corps         : '{' DeclVars SuiteInstr '}' ;
SuiteInstr    : SuiteInstr Instr
              | ;
Instr         : LValue '=' Exp ';'
              | READE '(' IDENT ')' ';'
              | READC '(' IDENT ')' ';'
              | PRINT '(' Exp ')' ';'
              | IF '(' Exp ')' Instr
              | IF '(' Exp ')' Instr ELSE Instr
              | WHILE '(' Exp ')' Instr
              | IDENT '(' Arguments ')' ';'
              | RETURN Exp ';'
              | RETURN ';'
              | '{' SuiteInstr '}'
              | ';' ;
Exp           : Exp OR TB
              | TB ;
TB            : TB AND FB
              | FB ;
FB            : FB EQ M
              | M ;
M             : M ORDER E
              | E ;
E             : E ADDSUB T
              | T ;
T             : T '*' F
              | T '/' F
              | T '%' F
              | F ;
F             : ADDSUB F
              | '!' F
              | '&' IDENT
              | '(' Exp ')'
              | NUM
              | CHARACTER
              | LValue
              | IDENT '(' Arguments ')'
              | '*' IDENT '(' Arguments ')' ;
LValue        : IDENT
              | '*' IDENT ;
Arguments     : ListExp
              | ;
ListExp       : ListExp ',' Exp
              | Exp ;
```

FIGURE 1.1 – Grammaire du langage TPC.

Chapitre 2

Manuel utilisateur

L'utilisation du compilateur s'effectue ainsi :

./bin/compil [FICHIER] [OPTIONS]

où FICHIER sera le fichier à traduire et compiler, ce qui génèrera un fichier **FICHIER.asm**. Compiler ce fichier asm pour générer un exécutable peut-être exécuté via la commande :

make nasm

Différentes options sont disponibles en argument après le fichier :

- -t Affiche l'arbre de la syntaxe abstraite.
- -s Affiche la table des symboles.

Si aucune option n'est fournie, la compilation sera effectuée et les avertissements/erreurs affichées, si des options sont fournies l'affichage sera effectué plus de la compilation. Il est possible de combiner les options ("-ts" par exemple).

Un script est disponible qui traduit et génère les exécutables des fichiers traduits pour les fichiers de test se situant dans le dossier **test**, celui-ci est utilisable avec la simple commande :

./script

Chapitre 3

Déroulement du programme

La traduction d'un fichier fourni est découpé en plusieurs étapes dans notre programme.

Premièrement, on effectue l'analyse syntaxique (= parsing) de notre fichier. Celle-ci correspond à vérifier que le fichier fourni est en accordance avec la grammaire du langage TPC. Pendant cette analyse est créé l'arbre de la syntaxe abstraite et est aussi vérifiée la présence d'une fonction **main** dans le programme, autrement il ne peut pas compiler et le programme est arrêté, avec un message d'erreur affiché sur le flux d'erreur standard.

Si l'analyse syntaxique n'a pas rencontré d'erreur, l'étape suivante est la création de la table des symboles en parcourant l'arbre de la syntaxe abstraite. La table des symboles est un moyen de stocker toutes les informations sur les différentes variables contenues dans le programme. Dans cette étape est vérifié qu'aucune variable n'est appelée sans être déclaré pendant le programme, si c'est le cas c'est une erreur et le programme est arrêté, avec un message d'erreur affiché sur le flux d'erreur standard.

Troisièmement, si la création de la table des symboles n'a pas rencontré d'erreur, avant de traduire le programme on vérifie que le typage des expressions est respecté. Les différentes erreurs de typage sont des erreurs de sémantique, telles que :

- Utiliser un pointeur en opérande pour une opération de comparaison ou une opération arithmétique.
- Utiliser un pointeur dans l'appel de la fonction print(), reade() ou readc()
- Retourner le mauvais type attendu par une fonction
- Appeler une fonction avec des arguments de types ne correspondant pas aux types attendus

Dans cette étape est aussi possible que des avertissements soient affichés, ceux-ci peuvent apparaitre pour de tels cas :

- Retourner un pointeur en fin de fonction alors qu'un type non pointé est attendu.
- Affecter un entier à une variable de type caractère.
- Appeler reade, readc sur un caractère, un entier (respectivement).

Enfin, si toutes les étapes précédentes sont retournées sans erreur, le fichier est valide et peut être traduit, c'est la dernière étape.

Chapitre 4

Choix algorithmiques et difficultés rencontrées

4.1 Choix algorithmiques

4.1.1 Fonctions

L'appel de fonction est en accordance avec les normes NASM : pour quelconque fonction, ses 6 premiers paramètres seront stockés dans les 6 registres NASM dédiés, tout paramètre suivant ceux-ci sera passé sur la pile. Un bloc d'activation est créé dans chaque fonction pour avoir l'aspect de variable locale à la fonction et globale.

4.1.2 Implémentation des booléens et comparaisons

Les opérateurs de comparaison et opérateurs booléens sont implémentés à l'aide de sauts conditionnels. Le code asm généré pour les opérateurs booléens fait l'évaluation paresseuse, par exemple pour l'opérateur la première condition est évaluée mais si celle-ci est fausse, l'intégralité de l'opération sera fausse : on empile donc un résultat faux (0 sur la pile).

L'opérateur de négation est appliqué en inversant les instructions d'empilement de résultat booléen. A l'aide d'une variable globale qui décrit si l'opérateur de négation est à appliquer sur l'opérateur que l'on évalue, on change l'instruction à générer. Dans l'exemple d'un opérateur , si la condition est vraie alors on empilerait 1 mais si l'opérateur de négation est utilisé, alors on génère l'instruction pour empiler 0.

4.2 Difficultés rencontrées

4.2.1 Restructuration

Suivant l'avancement du projet, une nécessité de séparation du déroulé de la compilation étapes à fait surface, notamment au moment de la vérification des erreurs. Initialement, nous avions pensés à effectuer la vérification des erreurs en parallèle de la traduction mais il semblait plus judicieux de générer un code ASM seulement si le fichier ne contenait pas d'erreurs, il nous à fallu intégrer la librairie *error_ check* et appeler sa fonction de vérification de présence d'erreurs avant la traduction.

4.2.2 Traduction

La traduction en code **NASM** fut une des plus grosses parties dans la réalisation de ce projet, qui à amené à la restructuration et de façon générale une plus grande appréhension du projet, du

fonctionnement d'un compilateur et même des structures déjà implémentée.

La traduction des pointeurs nous a été assez difficile, on n'a en effet réussi qu'à implémenter les affectations dans le même scope. En effet, si on a par exemple ce code : int *a, b, c ; b = 2 ; a = b ; c = *a ;

Tout d'abord, pour l'affectation "a = b ;" , nous empilons la variable a, puis l'adresse de b (récupérée dans la table des symboles), et on dépile les deux valeurs pour affecter l'adresse de b dans a.

Ensuite, pour l'affectation "c = *a ;", nous empilons la variable c, puis, "*a"qui correspond à la valeur de b. Ainsi, pour récupérer la valeur de b, nous regardons la valeur de a, qui est une adresse (celle de b), et on n'a plus qu'à récupérer la valeur présente à l'adresse trouvée.

Cependant, cette méthode ne fonctionne seulement quand les différentes variables affectées sont dans le même scope. Par exemple un pointeur déclaré dans une fonction f1, puis passé en argument d'une fonction f2 n'aura pas la valeur attendue si on souhaite l'affecter dans f2. Cela est dûe au fait que les paramètres récupérés par une déclaration de fonction sont empilés dans un bloc d'activation propre à chaque scope.