Université Paris Scalay

Faculté des Sciences

PROJET de COMPILATION

analyseur syntaxique et compilateur pour microGo

Présenté par :

Wang Marc — Pourchot Alistair

Groupe : LDD3-MAG-INFO

Introduction

Ce projet réalisé dans le cours de Compilation vise à réaliser un analyseur et interprète syntaxique, ainsi qu'un compilateur produisant du code MIPS pour le langage micro-go, un fragment (relativement petit) du langage Go.

Contents

1 Partie I : Analyse Lexicale et Syntaxique

La première étape de la compilation consiste à implémenter l'analyseur lexical (`mgolexer.mll`) et l'analyseur syntaxique (`mgoparser.mly`).

1.1 Analyseur Lexical (`mgolexer.mll`)

Le lexer est chargé de reconnaître et de convertir les séquences de caractères en unités atomiques appelées **tokens**.

Dans ce fichier, on distingue plusieurs étapes importantes :

- Tout d'abord : la **Tokenisation**. La tokenisation ça consiste en la reconnaissance des différents mots clés (if, for, func, etc), des identifiants, des opérateurs et des types de base (int, string, bool, nil)

- Il y a également la gestion des **commentaires** : il faut gérer différement les blocs "/* ... */" et les lignes de commentaire "// ... "

- Enfin, on s'occupe d'ajouter un **point-virgule** automatiquement lorsqu'un retour à la ligne est rencontré, avec comme condition que le dernier token lu avant ce retour à la ligne appartienne à une liste spécifique de tokens. Par ailleurs, nous avons observé qu'il nous était demandé de mettre un point-virgule uniquement lorsqu'un retour à la ligne était rencontré, cependant cela crée une erreur lorsqu'un fichier se termine sur la dernière ligne, sans retour à la ligne. Dans ce cas là, le point virugule automatique ne fonctionnais pas ; nous avons corrigé ce soucis.

1.2 Analyseur Syntaxique (`mgoparser.mly`)

Le parser construit l'AST en appliquant la grammaire du langage.

Il contient :

- La structure du programme : un programme doit commencer par "`PACKAGE main SEMI`" et il peut inclure l'import de "fmt". C'est également le parser qui gère les déclarations de structures "`TYPE ... STRUCT`" et de fonctions "`FUNC`".

- Il comporte également tout ce qui concerne les priorités et les règles d'associativité : la grammaire doit définir l'ordre d'évaluation des expressions.

- On va finalement y trouver toutes les instructions clés : les déclarations, les boucles, les expressions

Nous avons eu quelques soucis sur ces fichiers car, dans le sujet, nous avons cette règle dans notre grammaire :

$$\begin{aligned}\langle \text{instr\_simple} \rangle ::= & \langle \text{expr} \rangle \\ & \mid \langle \text{expr} \rangle (++ \mid --) \\ & \mid \langle \text{expr} \rangle^+ = \langle \text{expr} \rangle^+ \\ & \mid \langle \text{ident} \rangle^+ := \langle \text{expr} \rangle^+\end{aligned}$$

Qui nous a causé pas mal de soucis, puisqu'on ne savait pas faire la différence entre "=" et ":=" Pour essayer de corriger nos problèmes, on a commencé par utiliser un seul token "set" pour les deux et regarder le type de l'expression à gauche (car ident est un sous cas de expr). Cette méthode ne fonctionnait pas. Finalement, notre solution sera d'utiliser deux tokens : "set" pour "=" et "coloneq" pour ":=". Ceci étant fait, on aura tout de même un match pour nos deux derniers cas afin d'éviter des conflits S/R.

1.3 TypeChecker

Le Typechecker est le fichier le plus long et celui sur lequel nous avons passé le plus de temps du projet. Comme son nom l'indique, c'est dans ce fichier que l'on vérifie tout type de cohérence, que ça soit les cohérences de types, des instructions correctes, des types de retour d'une fonction correspondant, si une structure existe, si toutes les instructions d'une fonction mènent à un `return`, etc. On va également à travers ce fichier vérifier la présence d'une fonction main dans chaque fichier, le cas opposé menant à une erreur ; On traîte également la vérification que chaque variables initialisée dans une fonction est bien utilisée. Il nous a fallu du temps pour comprendre que nous devions renvoyer un environnement dans pratiquement chaque cas, car par exemple si nous faisons une initalisation d'une variable, celle-ci doit à présent être accessible dans notre environnement, et nous devons donc ajouter cette variable puis ensuite renvoyer cet environnement pour que cela fonctionne.

Gestion de l'Environnement et Passage de Contexte

La vérification de type repose sur le maintien de trois environnements distincts.

- `tenv` : Stocke les **variables locales** et leur type.

- `fenv` : Stocke les **fonctions** et leur signature (types des paramètres et types de retour).
- `senv` : Stocke les **structures** (`Struct`) et la liste de leurs champs.

Le principe de la **propagation d'environnement** est essentiel, car, après une déclaration de variable, l'environnement doit être mis à jour et renvoyé par la fonction de vérification d'instruction (`check_instr`) pour que la variable soit accessible aux instructions suivantes. Comprendre ceci nous a demandé beaucoup de temps, car au début nous n'avions pas pensé au fait de renvoyer un environnement, mais il se trouve que c'était la bonne solution.

Vérifications de Cohérence Fondamentales

La vérification s'étend à tous les éléments du langage :

La **cohérence des types** est gérée par la fonction type-expr, qui vérifie que les opérateurs sont appliqués à des types compatibles.

On gère les **appels de fonctions** en vérifiant que le nombre d'arguments ainsi que tous les types correspondent exactement à la signature dans fenv.

Il faut s'assurer que les structures référencées existent dans l'environnement des structures, et que la comparaison avec Nil n'est autorisée uniquement avec un type TStruct.

Quant à l'affecation, le vérificateur garantit que le type de l'expression à gauche est compatible avec celui à droite de l'opérateur.

Contrôles Sémantiques Strictes

Le typechecker applique des règles sémantiques plus larges, essentielles au comportement du programme :

Une analyse est menée pour identifier et signaler toute **variable** déclaré mais jamais utilisé dans le corps de la fonction.

Le typechecker doit s'assurer de la présence d'une fonction **main** dans chaque programme.

Il doit s'assurer également que **tous les chemins d'exécution** possibles se terminent par une instruction `Return`.

2 Partie II

Dans la deuxième partie, nous devions traiter l'aspect compilation et génération de code assembleur MIPS.

On a donc un fichier `mips.ml` contenant la traduction d'une opération vers un chaîne de caractère correspondant à l'équivalent en code MIPS, et dans notre fichier `compil.ml` nous avons implémenté la traduction des expressions et instructions du langage microGo vers ce code MIPS.

2.1 Principes du Compilateur et Gestion de la Mémoire

La compilation utilise le registre `$t0` pour contenir le résultat de chaque expression et l'environnement `Env` associe les variables à leurs **offsets** sur la pile (`$sp`).

Allocation des Données :

On traîte différement l'allocation selon le type de la donnée.

Les **chaînes de caractères** sont allouées dans la section data, et sont ensuite chargée dans $t0 par "`la`"

Pour allouer des **structures**, on utilise le syscall 9 afin d'obtenir de l'espace sur le tas. Enfin, l'accès aux **champs** de structure est basé sur un environnement global stockant l'offset de chaque camp.

2.2 Gestion des Fonctions et du Flux de Contrôle

Appels et Retours

La fonction `tr_expr` pour les appels de fonction gère la **sauvegarde de** `$ra` et l'empilement des arguments avant le `jal`. Le résultat de l'appel est récupéré depuis `$v0` vers `$t0`.

La principale difficulté réside dans la gestion du **retour de multiples valeurs**. Le compilateur :

1. Évalue `e1` vers `$v0` (via empilement/dépilement).

2. Évalue `e2` vers `$v1`.

3. Libère l'espace total alloué par les déclarations locales (via `stack_acc`) avant le `jr $ra`.

Déclarations Locales et Instructions

L'instruction `Vars` (`tr_seq`) alloue l'espace pour une **frame locale** sur la pile (`addi $sp $sp -frame_size`), met à jour l'environnement, puis libère cet espace à la fin. Le paramètre `stack_acc` est nécessaire pour que l'instruction `Return` puisse correctement désallouer la totalité de l'espace local avant de sauter à l'adresse de retour.

3 Tests

Il nous était fourni un ensemble de test très léger dès le début du projet. Nous avons donc ajouté, au fur et à mesure, des fichiers de tests personnels. Par ailleurs, dans les fichiers fournis, l'un d'entre eux ne contenait pas de fonction main, ce qui pose problème lors de l'analyse puisque nous levons une erreur lorsqu'un fichier ne contient pas justement de main. Nous en avons donc simplement rajouté un. Après ça, tous nos fichiers de tests sont corrects et passent sans soucis.

Pour transformer un fichier en fichier assembleur, on utilise la commande
$->$ `./mgoc.exe tests/nom_fichier.go`

(Note : il est possible qu'il y ait des problèmes de permissions lors de l'envoie des fichiers, si c'est le cas faire : `chmod +x *.sh`)

Puisqu'il y a beaucoup de tests, on a également conçu un script permettant de faire les analyses syntaxiques de tous les fichiers tests
$->$ `./TEST.sh`

Maintenant qu'on a tous nos fichiers MIPS, on peut à nouveau utiliser un script permettant de tous les executer en une fois
$->$ `./EXEC.sh`
Attention, nous utilisons l'installable spim pour executer nos fichier dans notre script.

Finalement, nos tests se constituent d'affectations multible, de liste chaînée, de fonctions à retour multiple, de structure pointant vers une autre structure, d'appel double. Voilà tout le contenu de notre projet de compilation.