Lundi 9 Janvier 2023

Université de Strasbourg

Rapport du projet de compilation

Contributeurs:
Charles JARDOT
Arnaud FILIPPI
Nathanaël DEROUSSEAUX-LEBERT
Thomas POIRRIER

Objectif: *Réaliser un compilateur pour le langage SoS donné. SoS est un langage impératif simple qui utilise une syntaxe et des fonctionnalités issues d'un sous-ensemble de langage shell unix (Sous-Shell). Ce compilateur produira en sortie du code assembleur MIPS qui devra s'exécuter sans erreur à l'aide d'un simulateur de microprocesseur MIPS.*

Introduction:

Ce rapport liste de manière exhaustive les fonctionnalités implémentées ou non dans le projet. Il rend compte du niveau d'aboutissement de celui-ci. Les fonctionnalités et implémentations réalisées les plus intéressantes seront détaillées par la suite.

Listes des fonctionnalités implémentées:

- La syntaxe et le lexique sont entièrement reconnues
- Argument du programme (– version, -tos (table des symbols), -o <name>)
- Fonction exit, echo, affichage des variables
- Toutes les opérations sur des entiers ou variables.
- Tableaux d'entiers
- Structures de contrôles et boucles (test, if, elif, else, while, until)
- Structure for des opérandes entières
- Jeu de test

Listes des fonctionnalités non-implémentés:

- Read
- Case (+ liste-cas + filtre)
- Fonctions et portées des variables
- Liste d'arguments
- Structures for sur des mots.
- Possibilité de mettre un tableau en entrée d'un for (`for id in ${id[*]}`)
- Possibilité d'utiliser des mots dans les tableaux.

Nous allons par la suite dans ce rapport, essayer d'expliquer brièvement ce qui a été fait, comment, et les potentielles limites.
Dans le code source est joint une batterie de tests. Pour exécuter les tests il faut taper dans le terminal "`sh tests.sh`"

Ces tests ont pour objectif de nous aider à savoir si notre code est correct. Cependant il se peut que certaines limites de nos implémentations n'aient pas été testées car nous n'avons pas pensé à un cas particulier bien précis.

SOMMAIRE

1) Fonctionnalités de base

1.1) Structure du projet

Le code est découpé en plusieurs blocs. Le premier étant la création d'outils pour manipuler des quads et la création d'une table des symboles, le second est la reconnaissance de la grammaire et la création des quads correspondant aux instructions. Enfin la dernière partie du code s'exécute une fois l'ensemble des quads créés et l'ensemble du texte reconnu. La liste contenant tous les quads est traduite au fur et à mesure en code MIPS.

Nous utilisons une liste globale de `Quad`.(quad de la forme `(op,op1,op2,res)`) Chaque `Quad` est généré par une fonction `GENCODE` propre à l'action désirée.

Concernant le MIPS nous avons créé des fonctions qui font des opérations simples. Lorsque nous générons du code MIPS, nous utilisons une combinaison de ces fonctions simples.

Nous avons aussi implémenté les fonctions traditionnelles pour procéder à des opérations sur des listes de quads telles que `complete, concat`, etc…

Pour simplifier l'usage des opérandes, nous avons aussi créé une famille de fonctions permettant de créer une opérande facilement. Par exemple, `to_operand_temp()` permet de créer une variable temporaire dans la table des symboles et de renvoyer, une opérande du bon type, pour le réutiliser par la suite dans des `GENCODE`.

1.2) Variables et affichage

Il est possible de créer des variables. Leur attribuer des valeurs, les manipuler, et afficher avec la commande "`echo`" leur valeur.

La table des symboles contient les informations sur chaque symbole. Il s'agit d'une hashtable, avec le nom comme clé, pour accélérer la recherche d'un id dans cette table. Chaque symbole à un type (`VAR`, `TAB`, `TEMP` ou `CONST`), un nom et une position dans la pile d'appel. Ensuite, certains types possèdent des attributs spécifiques, comme `TAB` qui possède la taille du tableau, ou `CONST`, qui contient la valeur de la constante.

2) Opérations, comparaisons et boucles conditionnelles

2.1) Opérations arithmétiques

Toutes les opérations sur les variables et opérandes entières sont implémentées avec leurs priorités respectives. Pour cela, on utilise des variables temporaires qui sont utilisées de la même façon que les variables classiques. Elles sont créées et placées dans la table des symboles. Chaque résultat d'opération à opérande unique ou multiple est placé dans une nouvelle variable temporaire. Toutes les opérations sont traitées avec la même fonction `gencode_operation` qui renvoie une opérande et traite ensuite séparément le cas du moins unaire du reste concernant la création du quadruplet.

On vérifie bien à chaque assignation et opération d'entier que la valeur ne dépasse pas les limites autorisées.

2.2) Comparaisons et Boucles conditionnelles

L'ensemble des comparaisons de `Opérateur1` et `Opérateur2` sont implémentées et fonctionnelles. Celles avec `Concaténations` ont été également implémentées. `OR` et `AND` sont également disponibles. L'agrégation de plusieurs conditions dans une condition est fonctionnelle. Concernant les boucles conditionnelles, `IF`, `ELIF`, `ELSE`, `WHILE`, `UNTIL` sont implémentées. `FOR` est implémentée à moitié. Seule la partie avec *Liste Opérande* est fonctionnelle. Nous avons rencontré des difficultés sur cette partie et n'ayant pas implémenté les fonctions, nous n'avons pas voulu nous pencher sur la gestion d'arguments.

`CASE` n'a pas été implémentée (Avec `filtre` et `liste-cas`)

L'implémentation se fait en utilisant des listes de `Quad` à compléter `.True`, `.False`. Nous utilisons également des listes `.Next` mais celles-ci sont moins nombreuses. Dans la manière dont nous avons codé des `GOTO` sont placés et complétés avec le bon branchement.

Les for fonctionnent uniquement avec des opérandes entières. Le fait de pouvoir y mettre des chaînes de caractères est une perspective d'amélioration. Aussi la possibilité de parcourir un tableau avec la syntaxe `for id in ${tab[*]}` ne fonctionne pas encore et serait à ajouter.

3) Tableaux

3.1) Déclaration et affectation

La déclaration est bien implémentée, elle est essentielle pour la suite des affectations. Pour cela, on ajoute dans la table des symboles l'id de ce tableau ainsi que toutes les données propres à un symbole. La taille du tableau est importante car on allouera quatre fois la taille du tableau dans la pile afin de pouvoir simuler chaque case de ce dernier.

Lors de l'affectation on vérifie que la case affectée existe bien, qu'on ne remplit pas une case non alloué sur la pile. Après la création d'une opérande tab et deux vérifications imposées par la grammaire, on crée une opérande `op2_str`. Cette opérande est alors remplie avec la valeur de l'opérande pointée par l'`Op_list`. Cela permet notamment de régler les problèmes d'affichage du contenu des cases du tableau (string ou entier). On vérifie également que l'opération `assign` est valide et que la case du tableau existe bien. Il ne reste plus qu'à générer le quad associé à cette assignation. On donne alors comme paramètre dans l'ordre la valeur de la case, son numéro et enfin sa position dans la table comme `result`.

Pour la partie mips, on se contente de remplir le `registre t2` avec la valeur de la case puis de placer la valeur du registre à sa place dans la pile. Tout cela en prenant compte le fait qu'un tableau prend plus de place de part sa taille dans la pile.

3.2) Liste-opérande, opérande et opérande-entier

On s'intéresse ensuite au cas des opérandes. Le cas `${id[*]}` n'est pas pris en compte par le compilateur. C'est pourquoi les tests liés à ce dernier renvoient `KO.`

Toutes les autres assignation à `opérande` et `opérande-entiers` ont été traitées. La fonction `gencode_tab_to_temp` permet de générer les quads relatifs à l'assignation des opérandes, entières ou non, à un tableau précédemment déclaré.

4) Tests de notre compilateur

Pour s'assurer que notre compilateur et ses fonctionnalités que nous avons implémentés fonctionnent comme attendu, nous avons créé une centaine de tests qui ciblent spécifiquement chacune d'entre elles. Ils se divisent en deux catégories: ceux qui testent et cherchent à provoquer des erreurs (de syntaxe par exemple) et ceux qui vérifient la justesse d'une sortie.

Le jeu de test est divisé en neuf catégories. Ce jeu peut être alors lancé via la commande `sh tests.sh` lorsqu'on est à la racine du projet. Le programme

exécute et affiche par catégorie chacun des tests avec la sortie `OK` ou `KO` dans le cas échéant.