Université de Strasbourg UFR de mathématiques et d'informatique

Présentation des capacités du compilateur SCALPA

2020-2021

Amir BOUAFIA

Adonis STAVRIDIS

Marion SAKUMA

Mika FILLEUL

Sommaire

Introduction

Dans le cadre des cours de compilation et de gestion de projets, nous avons effectué un compilateur pour le langage SCALPA. Dans ce rapport, nous présenterons les capacités de notre compilateur, ainsi que ses limites. Pour plus d'informations sur l'ensemble du projet, vous pouvez consulter la documentation Doxygen générée avec la commande suivante :

```
make doc
```

1. Déclaration et affectation de variables

D'abord le compilateur comprend les déclarations et affections de variables. SCALPA est un langage typé et il est donc essentiel de préciser le type d'une variable au moment de sa déclaration. Les différents types sont les entiers (int), les booléens (bool) et les tableaux (array). Ces variables sont stockées dans une table de symboles, qui crée une nouvelle entrée pour chaque variable au moment de sa déclaration et vérifie que l'affectation est de type équivalent.

Une variable peut être de nom quelconque mais elle doit commencer par une lettre majuscule ou minuscule et suivre par tout caractère alphanumérique et même le caractère ‘ et _.

Une variable est déclarée de la façon suivante :

```
var <variable> [ , <autreVariable> ] : <type> ;
```

Une affectation est donc faite ainsi :

```
<variable> := <valeur> ;
```

Un exemple de déclaration est le suivant :

```
program exemple
     var a, b : int;
     var flag : bool;
     begin
          a := 0;
          b := 1;
          flag := false;
     end;
```

2. Expressions arithmétiques

Le compilateur comprend les expressions arithmétiques avec les opérateurs +, -, *, / et ^, effectuant respectivement une addition, une soustraction, une multiplication, une division et une puissance. Il comprend également les opérations unaires et il est donc possible d'écrire -3 par exemple. Toute expression peut être entourée de parenthèses ( ).

Les expressions arithmétiques s'écrivent de la façon suivante :

```
<expression> <opérateur> <expression> ;
```

Un exemple d'expressions arithmétiques est le suivant :

```
program exemple
     var a, b, c, d, e : int;
     begin
          a := 5;
          b := -1;
          c := a + b;
          d := c * 2;
          e := 2 ^ c;
     end;
```

Le compilateur génère du code en sortie, tel que si une division par zéro tenait à s'effectuer, un message d'exception soit affiché et le programme s'arrête.

3. Expressions booléennes

Le compilateur SCALPA comprend également les expressions booléennes avec les opérateurs logiques and, or, not et xor ainsi que les opérateurs de comparaisons =, <>, <, <=, > et >=. Les opérateurs logiques effectuent des opérations uniquement sur des booléens alors que ces-derniers comparent des expressions arithmétiques. Une vérification de type a été mise en place pour s'assurer du bon typage au moment de la comparaison.

Les expressions booléennes s'écrivent de la façon suivante :

```
<expression> <opérateur> <expression> ;
```

Un exemple d'expressions booléennes est le suivant :

```
program exemple
    var a, b, c, d, e : bool;
    begin
        a := false;
        b := true;
        c := a and b;
        d := not c;
        e := d xor true;
    end;
```

4. Lecture et écriture

Deux fonctions du langage SCALPA permettent de gérer les entrées et sorties d'un programme. Grâce à la fonction read le programme peut lire une donnée d'un utilisateur et avec la fonction write il peut écrire des données dans un terminal. La fonction write peut aussi afficher des chaînes de caractères. Il faut noter que celles-ci ne peuvent pas être stockées dans des variables. La fonction read affecte à une variable la valeur passée en entrée en fonction de son type et la fonction write affiche également la valeur qui lui est passée en argument, en fonction de son type.

Les fonctions read et write sont utilisées de la façon suivante :

```
read <variable> ;
write <string | variable | expression> ;
```

Un exemple d'utilisation de ces fonctions est le suivant :

```
program exemple
     var a : int;
     begin
          write "input value of a:";
          read a;
          write "value of a is";
          write a;
     end;
```

5. Branchements conditionnels

Le compilateur permet d'effectuer des branchements conditionnels avec ce langage. Il est donc possible d'effectuer des opérations avec les opérateurs if, then, else ainsi que des boucles avec while, do. Il est possible d'effectuer autant de branchements souhaités. En effet, il est possible de mettre des boucles while dans des if et inversement.

Les branchements conditionnels sont utilisées de la façon suivante :

```
if <expressionBooléenne> then <instr> [ else <instr> ] ;
while <expressionBooléenne> do <instr> ;
```

Si les instructions sont plus qu'une, il faut les encadrer des mots-clés 'begin' et 'end'. Un exemple d'utilisation de ces branchements est le suivant :

```
program exemple
     var a, b : int;
     begin
          a := 0;
          while a < 4 do
          begin
               write a;
               if ( a = 0 or a = 2 ) then write "pair"
               else "impair";
               a := a + 1;
          end;
     end;
```

6. Gestion de tableaux

L'une des fonctionnalités les plus complexes du compilateur est la gestion de tableaux. Tout d'abord, il est possible de déclarer des tableaux de toutes dimensions de type int ou type bool uniquement. Ainsi dans un programme il est possible d'accéder aux valeurs de ce tableau mais aussi de les modifier. Au moment de sa déclaration toutes les valeurs sont mises à 0. Il est aussi possible d'utiliser read et write pour affecter ou afficher des valeurs de tableau.

Les tableaux sont à utiliser ainsi :

```
var <variable> : array[ <entier> .. <entier> , ... ] of
<type> ;
array[ <entierDim1> , <entierDim2> , ... ] := <valeur> ;
```

Un exemple d'utilisation des tableaux est le suivant :

```
program exemple
     var tab : array[-2..0] of int;
     var a : int;
     begin
          a := -2;
          while a <= 0 do
          begin
               tab[a] := a;
               write tab[a];
               a := a + 1;
          end;
     end;
```

Le compilateur vérifie si l'accès ou l'affectation à un élément du tableau est de bonne dimension, mais génère un message de segfault dans le code mips, si l'accès se fait en dehors des bornes du tableau.

7. Gestion de fonctions

Malheureusement, pour des raisons de temps, cette fonctionnalité n'a pas été implémentée, même si la table de symboles a été préparée pour les implémenter ainsi que la bonne gestion des scopes / portées des variables.

8. Arguments du compilateur

Le compilateur peut récupérer plusieurs options à son exécution. L'option 'version' permet d'afficher les contributeurs du projet, l'option 'tos' affiche la table des symboles lorsque la lecture du fichier passé en entrée a été achevée et l'option 'o' est utilisée avec un chemin pour indiquer le fichier de sortie. Le compilateur peut être appelé uniquement avec l'option 'version', sinon il doit toujours prendre en argument un fichier en entrée. Il vérifie également si le fichier en entrée est d'extension .p ou .scalpa. Si l'option 'o' de fichier de sortie n'a pas été indiqué alors le compilateur va générer le code par défaut dans un fichier qui s'appelle scalpaProgram.s. Aussi le compilateur peut générer par

lui-même tous les répertoires nécessaires pour le fichier de sortie, s'ils n'existent pas déjà.

9. Gestion des erreurs

Tout au long de la compilation, le compilateur s'arrête s'il trouve une erreur de syntaxe, de typage ou tout autre, et essaye de l'afficher dans la sortie d'erreur de la façon la plus compréhensible possible. Il affiche le token et la ligne à laquelle l'erreur est survenue, surtout si elle est d'aspect syntaxique. Donc le compilateur permet de détecter les erreurs assez facilement et d'indiquer où elles sont survenues.

10. Points intéressants

Notre compilateur peut présenter certains points intéressants qu'il serait nécessaire de mettre en valeur. Tout d'abord, l'implémentation de la table de symboles suit la logique d'une table de hachage ou l'accès est très rapide. Notre table de symboles suit la structure de la figure 1, ce qui nous permet de facilement accéder à tout symbole.

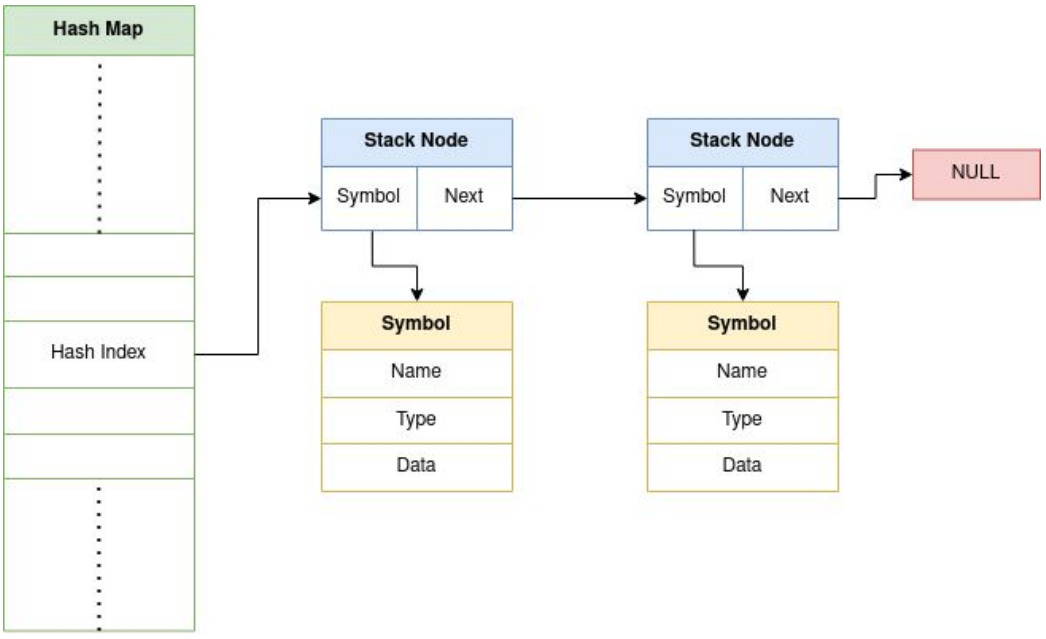

Figure 1 - Schéma de la table des symboles

Sinon le compilateur ne présente aucune fuite mémoire, en cas de succès. Toute la mémoire allouée sera désallouée lorsque le programme s'achèvera. Nous avons aussi créé de la documentation Doxygen, grâce aux commentaires du code, pour rendre plus claire la nécessité de chaque partie du code du projet, et afficher graphiquement comment le projet est construit.

Nous avons également tenté d'effectuer un minimum d'optimisation de code évitant de réécrire des instructions MIPS redondantes. Lorsqu'un bloc de code peut être réutilisé plusieurs fois dans un programme, on génère une fonction la première fois qu'un ensemble d'opérations doivent être effectuées, et elle est donc appelée lorsque ces mêmes opérations sont effectuées, plus tard dans le programme. Ceci uniquement permet d'enlever une dizaine de lignes minimum du code généré.

Enfin, nous avons essayé de rendre le code lisible et propre, en le structurant en plusieurs structures de données, plusieurs types et plusieurs fonctions. Cela nous a beaucoup aidé pour travailler en tant que groupe, mais aussi pour déboguer le projet.

Conclusion

En conclusion de ce projet, nous avons beaucoup réfléchi pour comprendre comment structurer le projet dans sa totalité, mais en organisant le travail en version incrémentale, de plus en plus complètes, nous avons réussi à développer la majorité des fonctionnalités attendues. Le travail est complet, à part pour les fonctions, et le programme fonctionne comme attendu.