Compilateur DECAF
Projet de compilation 2021-2022

Rayan LAJARGE - Ahmet Sefa ULUCAN
Massimo VENUTI - Alexandre VOGEL

10 janvier 2022

1 Utilisation

La commande **make** génère un exécutable nommé **decaf**. On utilise le compilateur de la manière suivante :

```
./decaf [options] <programme_decaf>
options : [-version] [-tos] [-o output_name] [-quad]
```

En plus des options obligatoires, nous avons ajouté une nouvelle option -quad qui permet d'afficher le code intermédiaire sur la sortie standard.

La commande **make doc** permet de générer la documention.

2 Capacités de notre compilateur

2.1 Règles sémantiques

Le compilateur vérifie explicitement chacune des règles sémantiques présentées dans le sujet et génère les messages d'erreur en conséquence.

2.2 Entrées/Sorties

Les quatre méthodes « *WriteInt, ReadInt, WriteBool, WriteString* » sont implémentées et fonctionnelles.

2.3 Vérifications dynamiques

Le compilateur procède aux vérifications dynamiques demandées pour générer une erreur lors d'une violation découverte à l'exécution.

2.4 Tests

Notre compilateur passe l'ensemble des tests du dépôt *dts* [1] avec succès. Nous avons également implémenté des tests unitaires qui couvrent environ 97% du code. Ces derniers se trouvent dans le dossier *test*. Quelques programmes DECAF supplémentaires se trouvent dans le dossier *decaf-programs*.

1. https://git.unistra.fr/alain/dts

3 Choix d'implémentation

3.1 Génération du code intermédiaire

Nous avons fait le choix d'une implémentation modulable en produisant du code intermédiaire avant de générer le code MIPS. Le code intermédiaire est ensuite traduit naïvement vers le code MIPS en retranscrivant les instructions à trois adresses en MIPS.

3.2 Table des symboles

La table des symboles se présente sous la forme d'une pile de tables de hachage qui gère les collisions par un chaînage des identificateurs. Par conséquent, la répartition et la recherche des identificateurs dans la table est optimale.

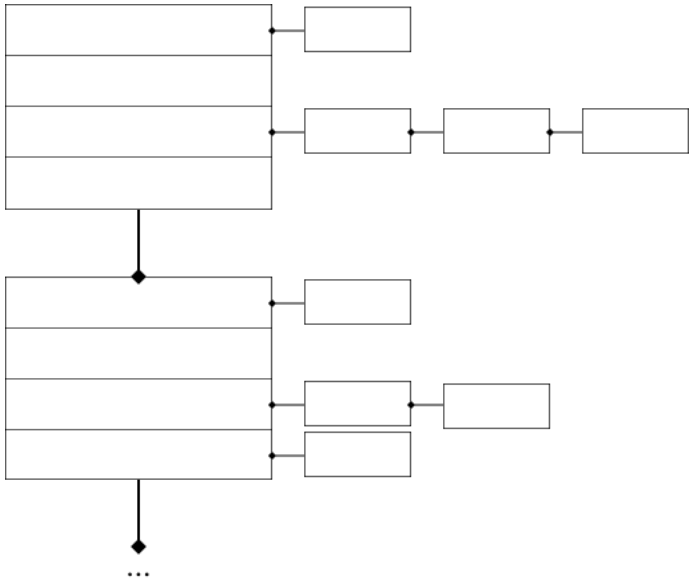

FIG. 1 – *Table des symboles*

3.3 Affichage des erreurs

En cas d'erreur, le compilateur met en évidence la source de l'erreur en exposant une indication graphique rouge sur les colonnes de la ligne concernée :

```
➜ ./decaf test.decaf
9:13: error: using void return value
   9 |         a = f();
     |             ^~~
```

FIG. 2 – *Exemple d'erreur*

3.4 Traduction du code intermédiaire en MIPS

Pour la génération du code MIPS, nous partons du principe que le code intermédiaire ne contient pas d'erreur, c'est-à-dire que nous n'effectuons aucune vérification sur le type des variable ou leurs existence ou encore la cohérence des *quads* [2].

Remarquons que nous nous servons de la table des symboles pour la traduction en MIPS, on ne stocke pas les informations nécessaires dans les *quads*. Nous avons préféré cette implémentation par soucis de simplicité.

3.5 Gestion de la mémoire

Chaque variable contient un *offset* qui permet d'obtenir son adresse dans la pile. Pour les variables globales, cet *offset* est fixé à -1 et son adresse définie dans le segment <data> sera identifié par un label. Les tableaux sont aussi identifiés par des labels dans le segment <data>.

Pour créer l'espace mémoire dédiée aux variables globales dans la section <data>, on part du principe que le premier *quad* est toujours un pointeur sur la table globale.

En MIPS, aucune distinction n'est faite par rapport au type d'une variable, tout est de la taille d'un *word*[3] (4 octets) que ce soit les entiers ou les booléens.

3.6 La méthode *main*

Contrairement aux méthodes classiques, la méthode *main* est soumise à une contrainte d'existence et de prototypage dans le programme DECAF. Les segments <text> du code MIPS commencent par générer et appeler la méthode main.

2. Représentation des instructions à trois adresses.
3. Mot mémoire.