Rapport du Groupe 1

Analyse syntaxique de la MTdv+

Calculabilité

Liza FRETEL, Kenza AHMIA, Alice WALLARD

17 Janvier 2023

1 Création de l'analyseur syntaxique de la MTdV+

Dans le cadre du projet collectif de la Machine de Turing de Del Vigna Plus, le groupe 1 se charge de la gestion de l'analyse syntaxique des programmes écrits en MTdV+. L'objectif est de vérifier qu'un programme en MTdV+ est valide et qu'il respecte les normes établies par l'ensemble des membres du projet lors du compte-rendu du 20 décembre 2023.

1.1 Fonctionnement de l'analyseur syntaxique de la MTdV+

L'analyseur syntaxique est organisé en 4 étapes, qui sont 4 fonctions du fichier TokenizeTSplus.py.

1. clean_lines() : cette fonction découpe le programme en lignes puis en tokens et nettoie les manques d'espaces éventuels entre des parenthèses et des opérateurs. Elle supprime les commentaires. Elle retourne la liste des lignes du programme associées à leurs numéros de lignes d'origine dans le programme. Les lignes serviront à l'affichage des messages d'erreur.

2. make_tokens() : il s'agit d'une fonction qui boucle sur chaque ligne du programme et essaie de déterminer de quel type d'instruction il s'agit. S'il y a un '=' seul, la fonction make_tokens sait qu'il s'agit d'une opération d'affectation ; et s'il y a '==', elle sait qu'il s'agit d'une opération de comparaison, ou un test. make_tokens() va ensuite appeler les fonctions qui vérifient la bonne formation des opérations, des affectations et des tests. Un test est constitué d'une opération à gauche, d'un signe ==, et d'une opération à droite. Ainsi, la fonction check_test() appelle deux fois check_operation(). Une affectation est constituée d'un nom de variable à gauche, d'un signe '=' et d'une opération à droite. L'affectation appelle une fois check_operation(). Les autres types d'instructions, comme *boucle, si,* etc, vont être tokenisées sur les espaces et chaque token sera traité un par un. La fonction make_tokens() ajoute la plupart des informations qui seront présentes dans le fichier de sortie .tsv pour chaque token : numéro de ligne, token, type du token, numéro de l'instruction, type de l'instruction et position dans l'opération (dans le cas de l'affectation ou du test).

3. check_structure() : cette fonction est quasi identique à l'analyse de la machine de Turing de Del Vigna. Elle vérifie la structure du programme, la bonne fermeture des si et des boucles, la présence du mot-clé fin dans les boucles et la présence du dièse à la toute fin du programme. De plus, elle ajoute aux tokens l'information « scope_boucle », qui est le numéro d'instruction de sortie de boucle pour tous les tokens qui sont à l'intérieur d'une ou de plusieurs boucles imbriquées. Cette information sera utile afin de savoir quand une variable cesse d'exister et elle peut être supprimée par les gestionnaires de mémoire et de variable.

4. write_tsv(): c'est la fonction du script principal qui écrit le fichier .tsv à partir de la liste des tokens. Pour chaque token, écrit : l'indice du token, le numéro de ligne dans le fichier d'origine, le token, le type du token, le numéro d'instruction, le type de l'instruction, la position dans l'instruction (G, M ou D) et le « scope_boucle ». Si « scope_boucle » est absent, parce que le

token ne fait pas partie d'une boucle, la fonction ajoute un – à la place afin de conserver le bon nombre de champs dans chaque ligne du tsv.

1.2 Conception de l'analyseur syntaxique de la MTdV+

La conception s'est effectuée avec le groupe, lors de la réunion du 13 décembre 2023. Nous avions déjà l'idée de générer un fichier TSV contenant le résultat de l'analyse et l'idée d'afficher les messages d'erreur. Nous n'étions pas encore sûres de quels devraient être tous les messages d'erreur, puisque c'est en programmant que nous les découvrions au fur et à mesure que nous en avions besoin.

Nous savions déjà qu'une grosse partie de l'analyse syntaxique de la MTdV+ proviendrait de l'analyseur syntaxique de la MTdV. Il a fallu imaginer cependant quelques règles syntaxiques autorisées et non autorisées et écrire quelques fichiers MTdV+ valides et non valides afin de pouvoir tester ces règles. Ces fichiers ont été placés dans TSplus/valides et TSplus/erreurs. Dans TSplus/erreurs/structure, il y a des fichiers TSplus dont l'erreur est structurelle, c'est-à-dire que ce sont des types d'erreurs qui peuvent apparaître dans un programme MTdV.

Nous avons ainsi défini plusieurs règles. Par exemple, les valeurs autorisées sont les entiers positifs allant de 0 à 29. Les opérateurs autorisés sont +, *, = et ==, opérateurs d'affectation et de test d'égalité. Nous aurions aimé ajouter –, < et >, mais l'implémentation semblait difficile pour le groupe 4 en charge de l'écriture du programme MTdV. Nous avons autorisé les noms de variable qui respectent la regex : ([a-z]|[A-Z]|_)([0-9]|[a-z]|[A-Z]|_)* .

1.3 Défis rencontrés

problème 1 :
Les valeurs 0 et 1 peuvent être soit de type valeur, soit de type MTdV. Comme nous avons décidé d'autoriser la commande 0 et 1, il a fallu désambiguïser s'il s'agit d'instructions ou de valeurs en regardant s'ils apparaissent dans une opération ou non.

problème 2 :
La récupération des informations de chaque token et l'ajout de celles-ci au travers des fonctions est relativement complexe. Il fallait garder en mémoire de nombreuses informations, notamment sur les tokens déjà traités par l'algorithme afin de faire une « stack ». La complexité de l'algorithme est un défi en soi.

problème 3 :
Nous avons eu un petit problème de communication avec le groupe 2. En effet, nous ne savions pas si c'était notre rôle ou le rôle du groupe 2 que de savoir quand une variable n'est plus appelée et doit être supprimée. Nous avions ajouté « scope_boucle » pour aider à savoir quand une variable dans une boucle doit être supprimée. Finalement, le groupe 2 s'est chargé de cela. C'était plus logique qu'ils s'en chargent puisqu'au sein de leur script existant, ils créent un dictionnaire des noms de variables. Ainsi, ils avaient déjà toutes les structures de données

nécessaires à l'implémentation de cet algorithme. Si nous avions eu à le faire, nous aurions dû ajouter une cinquième fonction tout en bas du script qui lit token par token et enregistre les informations dans un dictionnaire python, alors que le groupe deux fait déjà cela en lisant notre fichier tsv.

1.4 Répartition du travail

L'objectif du travail étant d'adapter l'analyseur syntaxique de la MTdV pour la MTdVplus, nous avions déjà une analyse récursive de la syntaxe d'un programme MTdV avec l'affichage des messages d'erreur, programmé au cours du semestre. Ainsi, le plus gros du travail à faire résidait dans la vérification de la syntaxe des opérations, des tests et des affectations. Nous avons chacune programmé une fonction parmi check_operation(), check_affectation() et check_test(), puis nous les avons mises en commun.

2 Gestion de l'intégration

Dans le cadre du projet collectif de la Machine de Turing de Del Vigna Plus, le groupe 1 se charge de la gestion de l'intégration du code. Il s'agit de faire en sorte que toute la compilation du programme MTdV+ vers MTdV soit facile à mettre en place pour l'utilisateur.

2.1 Rôle de l'intégrateur du code

Compte tenu de la structure de l'arborescence du projet, nous avons choisi de créer un fichier compile.sh à la racine du projet. Ce fichier a accès aux dossier scripts_python, TSplus, TSV et TS, qui sont les dossiers où sont rangés nos scripts TSplus utilisés pour nos tests, les scripts TSV générés par l'analyseur syntaxique et les fichiers compilés, au format .TS.

2.2 Défis rencontrés

problème 1 :
Les affichages des différents scripts (print) n'étaient pas pertinents pour la compilation. L'utilisateur a juste besoin de savoir s'il y a des erreurs dans son fichier TSplus, et si oui ou non la compilation a abouti. Ainsi, nous avons demandé aux autres groupes de ne rien afficher sur la console et de ne garder que les affichages de l'analyseur syntaxique.

problème 2 :
Lorsque l'appel d'un des scripts n'a pas fonctionné, il faut arrêter la compilation. Heureusement, il existe une variable en bash qui vérifie l'état de sortie (0 ou 1) d'un fichier exécutable : $?. Si l'analyseur syntaxique a détecté des erreurs de syntaxe, il fait toujours sys.exit(1), ce qui permet au compilateur de s'arrêter.

2.3 Répartition du travail

Pour la gestion de l'intégration, Liza s'est occupée de l'écriture du fichier compile.sh, et Kenza et Alice de tester si le processus de compilation est fluide.