I GDP

Globalement, chacun des trois membres du groupe a travaillé une trentaine d'heures sur le projet. Cinq réunions ont été réalisées (deux sous forme de TPs, et trois en *stand-up meeting*), pour se répartir les tâches, faire le point sur les prochains plans d'action, ... Dans la mesure où une messagerie très active a été mise en place au sein des membres du groupe, les problèmes plus mineurs et les répartitions moins importantes n'avaient pas lieu d'engendrer une réunion.

Voici la répartition des responsabilités de chacun :

- Ana : construction des tables des symboles des blocs de contrôle et conception des fonctions génératrices de structures `if`, `if-else` et `for`, et des protocoles E/S des fonctions ;
- Amine : conception de la fonction `dfs_type_check` (voir Section II), conception de la fonction génératrice de code pour les opérations (arithmétiques, de chaînes de caractères, de liste, booléennes) et du protocole d'affichage (`print` et de toutes les possibilités d'arguments) ;
- Antoine : construction des structures globales des fichiers, construction des tables des symboles des fonctions, de la fonction génératrice de code pour les appels de fonctions, et des ré-élagages de l'AST pour faciliter les générations de code.

II Table des symboles

Essentiellement, nous avons généré les tables des symboles comme un arbre unidirectionnel, chaque table ayant connaissance des tables englobantes. Chacune contient tous les symboles qui sont déclarés dans sa portée, avec un déplacement associé (négatif pour des paramètres, et positif pour les variables locales), ainsi qu'un type statique. Ce dernier, pour les paramètres, est mis par défaut à `unknown`. La fonction `dfs_type_check` permet de vérifier la cohérence des types d'une opération : elle soulèvera une erreur si les opérations demandées ne sont pas possibles vis-à-vis des types statiques. Dans le cas de paramètres, s'ils sont encore typés `"unknown"`, un type leur sera rétro-propagé pour refléter le type nécessaire au bon déroulé de l'opération dans laquelle ils sont impliqués en premier.

III Implémentation des structures de contrôle

III.1 Implémentation des `if` et if-else

Principe général

Dans un premier temps on évalue la condition dans l'instruction `if` en utilisant la même fonction qui génère/évalue des expressions.

- Si la condition est vraie, cette fonction renvoie 1 et le programme exécute les instructions à l'intérieur du bloc "if".
- Sinon, la condition renvoie un entier différent de 1 et la condition sera considérée comme fausse. Le programme passe au bloc "else" s'il existe, sinon on continue l'exécution du programme.

Ce résultat est stocké temporairement dans un registre, qui est ensuite comparé à 1 afin de déterminer quelle branche exécuter.

Concrétement :

1. le registre `rax` (qui contient l'évaluation de l'expression) est comparé à 1. Si la condition est fausse, on saute directement au label `else_...` ou `end_if_...` en utilisant l'instruction `jne` ;
2. si la condition est vraie, le code du bloc `if` s'exécute normalement et à la fin de ce bloc, un saut `jmp` permet d'éviter le bloc `else` ;
3. si la condition est fausse et si un bloc `else` existe **directement après le if**, l'exécution continue au label `else_...`, où le bloc `else` est exécuté ;
4. tous les chemins rejoignent le label `end_if_...` après l'exécution du bloc concerné.

Remarque (Gestion des labels)
Pour chaque structure de contrôle, les labels en assembleur sont nommés de façon systématique en combinant le type du bloc, le numéro du bloc dans le code et le numéro de ligne dans le code. Par exemple `else_0_5` marqué le début d'un bloc else, qui est le premier bloc `else` dans le code et qui est situé à la ligne 5. Cette convention permet de se repérer plus facilement et permet d'éviter les doublons éventuels.

III.2 Implémentation des blocs "for"

Les boucles `for` sont traduites sous forme de **fonctions** en assembleur, avec protocole d'entrée/sortie usuel. Chaque appel à une boucle correspond donc à un appel de fonction. Pour bien gérer la gestion des boucles imbriquées, on a fait le choix de considérer que les paramètres implicites de la fonction for sont :

- le compteur d'itération `i` ;
- l'élément courant de la liste `list[i]`.

Ces deux valeurs son stockées dans la pile avant l'appel du `for`comme les paramètres classique. Le compteur `i` est à 0 et mis à jour à chaque itération et l'élément courant `list[i]` est mis à jour à chaque tour de boucle. Ainsi, les blocs `for` peuvent être réutilisés, copiés ou déplacés dans le programme sans dépendre de variables globales ou des registres qui sont susceptibles d'être réécrits.

Structure générale

1. Avant d'appeler la boucle, on empile les deux paramètres implicites nécessaires à son exécution : le compteur d'itération `i` et élément courant de la liste.
2. L'exécution du bloc `for` se fait via un appel standard à la fonction correspondante, utilisant l'instruction `call for_...`.
3. À l'entrée de la fonction, on réalise le protocole habituel : sauvegarde de la base de pile (`rbp`) et allocation d'un espace mémoire local sur la pile (`sub rsp, ...`).
4. En début de boucle, on compare la valeur du compteur `i` à la taille de la liste (stockée dans la section ".data"). Si `i` est supérieur ou égal à cette taille, un saut conditionnel (`jge`) permet de sortir de la boucle en se dirigeant vers le label `for_end`. Sinon, l'exécution du corps de la boucle continue normalement.
5. À la fin de chaque itération, le compteur est incrémenté, puis un saut inconditionnel (`jmp`) renvoie au début du test de boucle pour lancer la suivante.

Remarque : Les labels sont gérés de la même manière que pour les blocs `if` et `else`.

Remarque sur ces deux parties : Pour assurer une bonne gestion des variables dans les différentes structures de contrôles, on a fait le choix de créer des **tables de symboles séparées** pour chaque bloc. Pour les boucles `for`, en plus de la table de symboles propre au bloc, on applique une allocation dynamique des paramètres implicites (le compteur `i` et l'élément courant `list[i]`) sur la pile. Ces paramètres sont placés à des offsets négatifs pour ne pas interférer avec les variables locales et leur déplacements (ils sont donc considérés comme des "paramètres" de la boucle, comme on considérerait les paramètres d'une fonction classique).