Projet Programmation : Partie 1

Pour ce projet de programmation, j'ai décidé de séparer le processus de compilation en 2 étapes successives : dans un premier temps, l'arbre de syntaxe abstraite donné en entrée est analysé de sorte à déterminer quel est le code assembleur qui doit être écrit, et le code en question n'est généré qu'ensuite. Cela me permet, en particulier, d'écrire dans mon fichier assembleur de manière non linéaire, afin de déclarer au début du fichier les variables globales et les chaînes de caractère, et ce quel que soit l'endroit où elles sont lues pour la première fois dans le code C.

Évaluation des expressions La règle principale pour l'évaluation des expressions est que lorsqu'une expression est évaluée, sa valeur finale doit toujours être contenue dans le registre `%rax`. Ainsi, pour une affectation de la forme `x =` e`;`, où e est une expression quelconque, il suffit d'évaluer e récursivement puis d'écrire dans le code assembleur `movq %rax,` a, où a désigne l'adresse de `x`. Cette restriction permet donc des évaluations récursives assez simples, bien que l'assembleur généré ne soit souvent pas optimal.

Pour les expressions faisant appel à plusieurs expressions, comme les opérateurs binaires typiquement, on commence par évaluer la deuxième expression puis on l'empile pour être certain de ne pas l'écraser. On évalue ensuite la première expression, puis on dépile la deuxième dans un registre callee saved (`%r10` dans la plupart des cas). A la fin de cette opération, la pile est inchangée par rapport à la situation initiale et les expressions à considérer se trouvent dans des registres connus, ce qui permet d'effectuer sereinement les opérations souhaitées.

Environnements et compteurs Les adresses des variables locales et des arguments des fonctions sont stockées par le compilateur dans un environnement `rho`, qui se présente sous la forme d'une liste de couples (nom, adresse). Cet environnement est spécifique à chaque déclaration de fonction et en particulier ne contient pas de variable globale, puisque les adresses de celles-ci sont de toute façon toujours de la forme x`(%rip)`, où x est le nom de la variable. En plus de cet environnement local, il y a un environnement global qui répertorie les labels des chaînes de caractère et une liste qui contient les fonctions renvoyant des entiers sur 64 bits, afin de gérer les appels à des fonctions extérieures. Deux compteurs sont également utilisés pour générer les labels liés aux chaînes de caractère (de la forme `.LC`i) et aux instructions conditionnelles (de la forme `.L`i). Pour les environnements globaux et les compteurs, des références ont été utilisées, bien que l'usage des références en Caml soit normalement interdit par la convention de Genève.

Écriture dans le fichier assembleur Afin d'écrire de manière non linéaire dans le fichier assembleur, j'utilise un tableau de chaînes de caractères `tab`, dont je concatène les cases une fois celui-ci rempli. Pour le remplissage de ce tableau, je souhaitais avoir le luxe d'utiliser des formats (`%s`, `%d`...) et ma fonction d'écriture typique se présente donc sous la forme `Printf.ksprintf (add` i`)` s a_0 $a_1 \dots a_n$, où s est une chaîne contenant des `%s`, `%d`... et les a_i sont les chaînes/entiers par lesquels il faut remplacer ces `%s`/`%d`. La fonction `ksprintf` du module `Printf` permet de créer la chaîne de caractère souhaitée

s' et d'exécuter la fonction `add` i s', qui ajoute la chaîne s' à la case i du tableau `tab`. J'ai essayé de factoriser cette fonction en une fonction `let write i = Printf.ksprintf (add i)`, mais on perd alors la possibilité de placer un nombre variable d'arguments, ce qui est l'un des grands intérêts de la fonction `ksprintf`. Je me suis donc contenté de cette écriture non-synthétique, que l'on retrouve un peu partout dans le code Caml.