TP n°1

Machines à pile et compilation

Le but de ce TP est d'étendre le code développé en cours, qui peut être trouvé sous le nom `myrte.ml` sur la page du cours. On appelle A l'accumulateur de la machine virtuelle et S sa pile. On rappelle que ses instructions sont:

push empile le contenu de A sur S

consti n remplace le contenu de A par n

addi dépile un élément n de S, remplace A par $A + n$

andi dépile un élément n de S, remplace A par 0 si $A = n = 0$, par une valeur quelconque $\neq 0$ sinon

eqi dépile un élément n de S, remplace A par 1 si $n = A$, 0 sinon

Exercice 1 (Conditionnelle)**.** On souhaite étendre le langage Myrte avec une construction conditionnelle **if** ... **then** ... **else** .... Sa sémantique est: "la valeur de l'expression **if** E_1 **then** E_2 **else** E_3 est la valeur de E_2 si la valeur de E_1 est **true**, ou la valeur de E_3 si la valeur de E_1 est **false**."

1. Étendre le type des expressions avec un constructeur If
2. Étendre et tester la fonction `interp` avec ce nouveau cas
3. Étendre la fonction de typage `check` (on considère que E_2 et E_3 s'évaluent toujours vers des entiers)
4. Ajouter l'instruction suivante à la machine virtuelle:

   branchif n saute n instruction en avant si $A \neq 0$; n'a pas d'effet sinon
5. Étendre la fonction de compilation `compil` et de (dés)assemblage de façon à compiler la nouvelle construction

Exercice 2 (Macros)**.** En plus de **branchif**, rajouter les instructions suivantes à la machine virtuelle:

acc n copie dans A le contenu du n-ème élément de S (en partant de l'élément 0 qui est le sommet de S)

assign n copie A dans la n-ème case de S

Puis, implémenter les macros suivantes, c'est-à-dire des fonctions OCaml qui renvoient des listes d'instructions ayant l'effet désiré sur la mémoire:

1. `goto` n saute n instructions en avant, de façon inconditionnelle. (les instructions qui la composent peuvent modifier A)
2. `branchifnot` n saute n instruction en avant si $A = 0$; n'a pas d'effet sinon. (les instructions qui la composent peuvent modifier A).

3. `pop` dépile l'élément au sommet de la pile et le place dans A.

4. `decr` n décrémente le contenu de la n-ème case de S. (attention à ne pas modifier l'accumulateur)

5. `swap` échange les deux premières cases de S (attention à ne pas modifier l'accumulateur)

Exercice 3 (Multiplication)**.** On souhaite maintenant étendre le langage Myrte avec une multiplication d'expressions entières $\ldots \times \ldots$. Sa sémantique est celle de la multiplication des entiers. Notez que la machine virtuelle n'a pas d'instruction **mult**; pour compiler $E_1 \times E_2$, on utilisera l'instruction **branchif** avec un nombre négatif d'instructions pour implémenter une boucle qui calculera $E_2 + \ldots + E_2$ (E_1 fois). Attention, le code correspondant à l'expression $E_1 \times E_2$ ne devra calculer E_1 et E_2 qu'une seule fois, et sa taille ne devra pas dépendre de la valeur de E_1 ou de E_2.

1. Étendre le type des expressions, les fonctions `interp` et `check` en conséquence

2. Étendre la fonction `compil` pour compiler la nouvelle construction. Le code correspondant à $E_1 \times E_2$ commencera par placer les valeurs de E_1 et E_2 au sommet de la pile, puis la valeur 0 qui constituera le résultat notre addition, puis entrera dans une boucle: tant que la troisième case de la pile (initialement la valeur de E_2) est différente de zéro, on ajoutera la valeur de E_1 au résultat, puis on décrémentera la valeur de la troisième case. A la fin de la boucle, on restaurera le contenu de S et on placera le résultat dans A.

3. L'expression -2×3 est-elle bien typée? Quelle est sa valeur vis-à-vis de `interp`? Quel est le résultat de l'execution de son code? Mêmes questions pour 3×-2.