TD3 : Allocation mémoire

Langage `E := cte | id | let id = E in E | id(E,E) | (E)`

Exercice 1 **Génération d'assembleur** Proposer une fonction qui lit une expression e et produit du code MIPS correspondant à l'évaluation de e stockée dans un registre `r`. Votre fonction pourra être écrite en `Ocaml`, pour l'AST représenté par le type suivant :
`type ast = Cte of int | Var of string | Let of string * ast * ast | Call of string * ast * ast`.
On supposera que l'on dispose d'une fonction `get : string -> int` qui indique pour une variable le décalage de la case de la pile où elle est stockée (par rapport au pointeur de cadre).
On dispose également d'une fonction `label : string -> string` qui associe à chaque fonction le label menant à son code. Les fonctions lisent les deux arguments positionnés sur le sommet de la pile au moment de leur appel.
Dans ce langage, les fonctions sont prédéfinies, donc vous n'avez pas à vous soucier de la sauvegarde et restauration du cadre d'activation, on considère que les fonctions retournent à l'appelant après avoir stocké leur valeur de retour dans `$v0`.
Les graphes de flot de contrôle pour ces codes ont-ils des propriétés spécifiques ?

Exercice 2 **Affectation sur pile** Considérez le code suivant :

```
let x = 1 in
 let t = let u = 2 in plus(u,3) in
  f(let y = h(2,t) in g(y,x), let z=4 in g(z,z))
```

Considérez le code MIPS généré par votre fonction, et considérez une affectation pour `get` utilisant :

- Trois cases mémoire sur la pile
- Deux ?

Exercice 3 **Représentation intermédiaire** Plutôt que d'utiliser explicitement la pile, on passe par un langage avec des pseudoregistres.
On considère le langage intermédiaire suivant :
`type v = Var of string | Stack of string`
`type iexpr = Icte of int | Ivar of v | Icall of string * v * v`
L'idée est de transformer l'expression en une suite d'affectations basiques de la forme `v := t`. `v` est soit une variable du programme source, soit un temporaire (un emplacement de pile anonyme). On se donne une variable `r` dans laquelle on stockera la valeur des expressions traduites. Si nécessaire, on créera de nouvelles variables `ri,rj` pour les calculs intermédiaires.

1. Donner la suite d'affectations pour le code de l'exercice précédent.
2. Indiquer en chaque point du programme l'ensemble des variables vivantes, sans prendre en compte les variables de pile `ri`.

3. Écrire une fonction `translate : ast -> (v * iexpr) list` qui transforme une expression en liste d'affectations pour cette représentation intermédiaire. On considère donnée une fonction `fresh : unit -> v` produisant une nouvelle variable de pile à chaque appel.

Exercice 4 **Affectations** On considère `l` une suite d'affectations produite par la fonction `translate` de l'exercice précédent.

1. Écrivez une fonction `live : (v * iexpr) list -> (v * iexpr * (v set)) list` qui étiquette chaque instruction du programme par l'ensemble des variables vivantes *au début* de cette dernière Pour cette fonction on suppose donné le type `set` avec les opérations `empty, add, remove`.
2. Proposez un algorithme qui affecte à chaque variable du code un entier correspondant au décalage sur la pile auquel sera alloué cette variable.