TD COMPILATION — FEUILLE 5

On s'intéresse à la génération de code des affectations du langage LX (voir feuille 4). On supposera que les instructions d'affectation sont représentées sous la forme d'un arbre abstrait défini par :

arbre_inst	→	**affect**(arbre_place, arbre_exp)	instruction d'affectation
arbre_exp	→	arbre_place	expression non constante
		| **cst**(integer)	constante avec sa valeur
arbre_place	→	**idf**(string, exp_type)	identificateur avec son nom
		| **index**(arbre_place, arbre_exp, exp_type)	opération d'indexation de tableau

L'information **exp_type** mémorise dans l'arbre le type d'une place.

Exercice 1. Construction de l'arbre abstrait

On construit les arbres abstraits à partir de la syntaxe de la manière suivante :

inst $\uparrow$ **affect**$(a1, a2)$	$\to$	place $\uparrow a1$ **:=** exp $\uparrow a2$
place $\uparrow$ **idf**(nom, tt)	$\to$	**idf** $\uparrow nom \uparrow tt$
place $\uparrow$ **index**$(a1, a2, tt)$	$\to$	place $\uparrow a1$ [exp $\uparrow a2 \uparrow tt$]
exp $\uparrow$ **cst**(val)	$\to$	**cst_ent** $\uparrow val$
exp $\uparrow a$	$\to$	place $\uparrow a$

On suppose que la vérification de type a été effectuée.

▷ **Question 1** *Donner l'arbre abstrait associé à l'expression* `t[x][a[x]]` *pour les déclarations suivantes :*

```
t : array [5] of array [4] of int;
a : array [3] of int;
x : int;
```

Exercice 2. Codage des expressions

On dispose d'une machine abstraite décrite par :
- un ensemble de registres pouvant contenir des entiers ou des adresses : notation R0, ...
- une mémoire dont les adresses sont numérotées de 0 à N
- quatre modes d'adressage : registre direct (notation Rm), registre indirect (notation (Rm)), immédiat (notation #d), et absolu (notation d).
- les instructions suivantes, où *ad* est une adresse quelconque :

  MULT ad Ri   ADD ad Ri
  LOAD ad Ri   STORE Ri ad

Par exemple l'instruction `x := 4` peut se traduire par la séquence d'instructions machine :

LOAD #4 , R0   STORE R0,0

si 0 est l'adresse associée à la variable x.

▷ **Question 1** *Donner une séquence de code pour l'expression ci-dessus. On désignera par ad_n l'adresse associée à un identificateur de nom n.*

▷ **Question 2** *Donner l'algorithme de codage des places et des expressions.* **On imposera dans un premier temps une évaluation de gauche à droite des opérations d'indexation**. *On utilisera les conventions suivantes :*

- *la procédure* `coder_place(a : in arbre ; r : out reg)` *produit du code qui positionne dans le registre* `r` *l'adresse mémoire de l'arbre de place* `a`.
- *la procédure* `coder_exp(a : in arbre ; r : out reg)` *produit du code qui positionne dans le registre* `r` *la valeur de l'arbre d'expression* `a`. *On ne s'intéresse ici qu'à des arbres de type* `int`.

On suppose disposer d'autant de registres que nécessaire. On utilisera la fonction `allouer return reg` *qui retourne un registre libre et la procédure* `liberer(r : in reg)` *qui libère un registre.*

▷ **Question 3** *Donner le calcul du nombre de registres nécessaire pour l'algorithme de génération de code proposé ci-dessus.*

▷ **Question 4** *On lève la restriction sur l'ordre d'évaluation des opérations d'indexation. Donner un exemple d'expression tel que l'évaluation droite/gauche nécessite moins de registres que l'évaluation gauche/droite. Reprendre les questions 1 et 2 pour traiter cette extension.*

▷ **Question 5** *Écrire la procédure* `coder_aff(a : in arbre)`, *où* `a` *est un arbre représentant une affectation entre entiers ou entre tableaux, qui produit du code réalisant l'affectation.*