Ce TP vous propose diverses manières d'étendre nos interprètes des langages IMP et FUN.

Exercice 1 (Paires (FUN))**.** Étendre la syntaxe abstraite et l'interprète du langage FUN avec une notion de paire. Il faudra étendre le type `expr` avec les trois nouvelles formes :

- `Pair` qui construit une paire à l'aide de deux expressions,
- `Fst` et `Snd` qui s'appliquent à une expression dont la valeur est une paire et qui extraient respectivement le premier et le deuxième élément de cette paire.

Le type `value` doit également être étendu pour inclure les paires de valeurs.

Exercice 2 (Booléens (FUN))**.** Modifier la syntaxe abstraite et l'interprète du langage FUN de sorte que :

- le langage contienne deux constantes booléennes `true` et `false`, et des opérateurs `&&` (conjontion), `not` (négation) et `=` (test d'égalité) ;
- les valeurs booléennes soient représentées par un nouveau cas dédié `| VBool of bool` plutôt que codées à l'aide des entiers.

L'opération de comparaison `<` ne devra comparer que des entiers, tandis que l'opération d'égalité `=` pourra comparer tous types de valeurs. Comme il est délicat de comparer deux fonctions, on ne considérera deux fermetures comme égales que si elles sont physiquement égales.

Exercice 3 (Interprète avec structures de données mutables (IMP))**.** Modifier l'interprète du langage IMP pour que l'environnement soit maintenant représenté à l'aide d'une table de hachage.

Exercice 4 (Branchement généralisé (IMP))**.** Étendre la syntaxe abstraite et l'interprète du langage IMP avec une notion branchement généralisé de type `switch/case`. On veut notamment ajouter une instruction `switch` comportant :

- une expression discriminante `e`,
- une séquence de branches de la forme `n: {s}` où `n` est un entier à comparer à la valeur de `e` et `s` une séquence d'instructions à exécuter le cas échéant,
- une séquence d'instructions par défaut, à exécuter si aucune branche n'a été sélectionnée.

Exercice 5 (Sorties de boucle (IMP))**.** Étendre la syntaxe abstraite et l'interprète du langage IMP avec deux instructions `break` et `continue`. Rappels :

- `break` met fin à l'exécution d'une boucle,
- `continue` met fin à l'exécution d'un tour de boucle,

et l'un et l'autre s'appliquent à la boucle la plus interne.

Indication. Voici deux pistes pour inclure dans l'interprète la sémantique de ces deux instructions :

- si vous le connaissez, vous pouvez utiliser le mécanisme des exceptions de caml,
- vous pouvez faire en sorte que les fonctions `execi` et `execb` renvoient une valeur indiquant la manière dont l'exécution s'est terminée, à choisir par exemple entre « normal », « `break` » et « `continue` ».

Il est également intéressant de comparer les deux versions.

Exercice 6 (Fonctions (IMP))**.** Étendre la syntaxe abstraite et l'interprète du langage IMP pour permettre la définition et l'appel de fonctions. On veut notamment ajouter :

- une instruction de définition d'une fonction,
- une instruction `return`,

— une expression d'appel de fonction.

Une fonction doit être définie par : un nom, une liste de paramètres formels (chacun donné par un nom), une liste de variables locales, et une séquence d'instructions à exécuter. Un appel de fonction est donné par le nom de la fonction à appeler et une liste de paramètres effectifs. Une instruction `return` s'applique à une expression donnant la valeur à renvoyer. On suppose qu'une fonction dont l'exécution termine sans instruction `return` renvoie 0.

Indication. Vous pouvez utiliser un deuxième environnement pour associer les noms des fonctions à leurs définitions.

Indication. Attention, l'instruction `return` peut apparaître n'importe où dans le corps de la fonction. Notez cependant que vous pouvez incorporer la gestion de `return` dans le mécanisme déjà en place pour `break` et `continue`, quelle que soit la stratégie adoptée les concernant.

Indication. Attention, les noms des paramètres ou des variables locales peuvent coïncider avec les noms d'autres variables du programme mais l'un et l'autre ne doivent pas être confondus. Il faut trouver un moyen de bien les isoler les uns des autres (vous pouvez néanmoins dans un premier temps ignorer cette difficulté).

Exercice 7 (Tableaux (IMP))**.** Étendre la syntaxe abstraite et l'interprète du langage IMP avec une notion de tableau. On veut notamment ajouter :

— une expression similaire au `[e]*n` de python, construisant un tableau de taille `n` dont chaque case contient la valeur de `e`,

— une expression `t[n]` d'accès à la case d'indice `n` du tableau `t`,

— une instruction `t[n] = e;` d'affectation d'une nouvelle valeur à la case d'indice `n` du tableau `t`.

Cette extension demande d'expliciter un type des valeurs des expressions IMP regroupant les entiers déjà utilisés et les tableaux. On pourra réutiliser les tableaux natifs de caml pour représenter les valeurs des tableaux IMP.

Exercice 8 (Représentation de la mémoire)**.** On peut naïvement représenter la mémoire adressable à l'aide d'un tableau Caml en faisant coïncider adresses et indices. Construire un tableau de longueur 2^{32} voire 2^{64} n'est cependant guère raisonnable.

On propose donc de découper la mémoire en plusieurs « pages » représentant chacune une plage d'adresses consécutives (de taille par exemple 4096), chacune représentée par un tableau, mais de n'allouer ces tableaux que pour les plages effectivement utilisées.

On utilisera donc une table de hachage contenant les pages utilisées. Plus précisément, étant donnée une adresse représentée par un entier caml, les 12 bits de poids faible donnent l'indice localisant cette adresse dans sa page, tandis que les autres bits servent de clé pour la table de hachage.

Écrire un module simulant ainsi une mémoire virtuellement aussi étendue que possible, et proposant des fonctions de lecture et d'écriture.