| Programmation IV | Série 12 | 1er juin 2004 |
|---|---|---|

## Exercice 1

Vous trouverez sur la page du cours une implantation de l'interpréteur LISP vu au cours. Le but de cet exercice est de le modifier pour ajouter du « sucre syntaxique ».

On appelle *sucre syntaxique* des parties de la grammaire d'un langage qui permettent d'exprimer des choses déjà existantes dans le langage, mais d'une autre manière, supposée plus agréable pour le programmeur.

Comme cela a été vu au cours, la gestion du sucre syntaxique est généralement confiée à une fonction de normalisation qui supprime ce sucre. Pour cet exercice vous devrez donc compléter la fonction `normalize` qui prend une expression contenant éventuellement du sucre syntaxique et en retourne une équivalente mais sans sucre.

Pour commencer, ajoutez une nouvelle forme de `def`, qui permette de définir une fonction de manière plus simple. L'idée est de transformer l'expression suivante :

```
(def (nom arg_1 ... arg_n) corps expr)
```

en l'expression suivante :

```
(def nom (lambda (arg_1 ... arg_n) corps) expr)
```

Cela permet par exemple de définir (et de tester) la fonction successeur de la manière suivante :

```
(def (succ x) (+ x 1) (succ 0))
```

## Exercice 2

Le but de cet exercice est d'ajouter à l'interpréteur une boucle de lecture–évaluation–impression (*read-eval-print loop* ou *REPL* en anglais).

Une telle boucle est au cœur de tout interpréteur interactif comme `scalaint`. Elle permet à l'utilisateur d'interagir avec le système en entrant des expressions et en examinant directement le résultat de leur évaluation.

Votre boucle devra signaler à l'utilisateur qu'elle est prête à accepter une expression au moyen d'une invite (*prompt* en anglais), qui pourra être p.ex. `lisp>`. Elle attendra ensuite que l'utilisateur entre une expression, l'évaluera, puis imprimera le résultat de l'évaluation.

Une interaction avec votre boucle ressemblera donc à ceci (le texte entré par l'utilisateur est souligné) :

```
lisp> (+ 1 3)
4
lisp> (cons 1 (quote ()))
(1)
lisp>
```

Pour réaliser cette boucle, vous aurez besoin d'accéder à des classes Java permettant de lire des données ligne après ligne. Les classes dont vous aurez besoin sont `BufferedReader`, `InputStreamReader` et `System`. Reportez-vous à la documentation officielle de Java pour plus d'information.

## Exercice 3

Écrivez en LISP une fonction `powerset` qui calcule l'ensemble des parties d'un ensemble, comme cela a été vu à la série 5. Vous aurez besoin pour cela de définir les fonctions `map` et `append` en LISP, ce qui constitue une excellente révision.

Testez cette fonction avec votre interprète, et vérifiez que vous obtenez le résultat escompté pour quelques exemples.

## Exercice 4 (optionnel)

Ajoutez une nouvelle forme d'expression conditionnelle, `cond`, permettant la définition agréable de chaînes de test. La syntaxe de cette forme est la suivante :

```
(cond (test_1 expr_1)
      (test_2 expr_2)
      ...
      (test_n expr_n)
      (else expr_else))
```

La signification de cette expression est la suivante : si le premier test est vrai, alors la première expression est évaluée et retournée ; sinon, si le second test est vrai, alors la seconde expression est évaluée et retournée, et ainsi de suite jusqu'au $n^e$ test ; si aucun des tests n'est vrai, l'expression suivant le mot-clef `else` est évaluée et retournée. La conditionnelle `cond` se traduit au moyen d'une chaîne de tests, de la manière suivante :

```
(if test_1
    expr_1
    (if test_2
        expr_2
        ...
           (if test_n expr_n expr_else)))
```

Indication : pensez à une solution récursive !