LICENCE INFORMATIQUE L3 — TP 14

(jpr), Nice, Programmation Applicative et Interprétation — *(hum...)*

Partie 1 : sans `call/cc`

Exercice 14.1 Utilisez `with-handlers` pour verrouiller complètement l'interprète MIPS du TP 11a ! On ne doit plus sortir de la boucle *REP* sur erreur :

```
> ($mips)
Mips Is Pseudo-Scheme !
? )
Mips read error !...
? (/ 1 0)
Scheme error : /: division by zero
? (+ x 1)
Mips error : Unknown identifier x
? quit
Program first, think later !
>
```

Exercice 14.2 Modifier l'évaluateur d'expressions arithmétiques [cours 11a page 11] pour qu'un appel à `capture!` ne provoque pas l'abandon du calcul, mais que celui-ci continue normalement, tout en capturant la continuation courante. Comme il peut y avoir plusieurs nœuds de racine `capture!`, gérez une *liste de continuations*, et écrivez au passage la macro `(push! x L)` empilant `x` dans la liste `L`, avec effet de bord sur `L` bien entendu. Utilisez `map` pour appliquer ensuite toutes les continuations empilées sur l'argument `1` par exemple…

Exercice 14.3 Utiliser *l'arpenteur général* d'arbre binaire d'expression [cours 14 pages 16-17] pour calculer, si l'arbre `A` est par exemple `'(+ (* 2 (/ 3 x)) (/ z -5))` :

a) le nombre de nœuds d'un arbre : `(visiter A …) → 4`

b) la première feuille qui est une variable : `(visiter A …) → x`

c) la dernière feuille de l'arbre : `(visiter A …) → -5`

d) la liste des feuilles de l'arbre : `(visiter A …) → (2 3 x z -5)`

e) le premier noeud de racine / : `(visiter A …) → (/ 3 x)`

Exercice 14.4 Reprendre le jeu des N dames sur un échiquier [cours 14 pages 18-19].

a) Terminez le programme en écrivant le prédicat `(ok? k i L)` testant si une dame en colonne `k` et ligne `i` est compatible avec la liste de dames `L` déjà placées sur les colonnes `1..k-1` *[on utilisera une valeur absolue pour exprimer que deux dames sont la même diagonale]*. Testez alors le programme qui fait afficher la première solution. Vérifiez qu'il n'y a des solutions qu'à partir d'un échiquier 4×4…

b) Faites afficher *toutes* les solutions pour 5 dames.

c) Combien y-a-t-il de solutions pour 8 dames (donc sur un échiquier normal) ?

Partie 2 : `call/cc`

Exercice 14.5 Que valent chacune des expressions suivantes au toplevel :

```
(+ 1 (call/cc (lambda (k) (* (k 4) 8))))
(+ 1 (call/cc (lambda (k1) (+ (call/cc (lambda (k2) (/ 5 (k2 8)))) (k1 10)))))
(+ 1 (call/cc (lambda (k1) (+ 3 (call/cc (lambda (k2) (k2 (k1 8)))) 10))))
(+ 1 (call/cc (lambda (k1) (+ 3 (call/cc (lambda (k2) (k1 (k2 8)))) 10))))
(+ 1 (call/cc (lambda (k1) (+ (call/cc (lambda (k2) (+ (k2 5) (k1 8)))) 10))))
```

Critiquez au besoin…

Exercice 14.6 Reprenez la macro `loop` [cours 14 page 26].

a) Elle contient un **bug profond** lié à l'hygiène, corrigez-le !…

b) Utilisez-la pour programmer impérativement l'inévitable factorielle `(fac n)`.

c) Ajoutez-lui le mot-clé `continue` de telle sorte que dans la portée d'une boucle `loop`, un appel à `(continue)` provoque une reprise immédiate à la première instruction du corps de boucle.

```
(let ((n ' ?))
  (loop (set! n (random 10))
        (printf "n=~a " n)
        (if (= n 5) (exitloop 'gagné) (continue))
        (printf "cette ligne n'apparaîtra jamais !")))
```

Exercice 14.7 *[Examen L3-INFO]*. Chargez la macro `amb` [cours 14 pages 27-31] que vous trouverez dans le fichier `amb.rkt`.… Il s'agit de programmer un problème classique présenté dans les livres sur le langage PROLOG pour illustrer la puissance du backtrack automatique de ce langage.

a) Programmez un prédicat `(tous-distincts? L)` prenant une liste `L` et retournant `#t` si et seulement si tous les éléments de `L` sont distincts.

b) Utilisez l'opérateur `amb` pour faire afficher une solution au puzzle arithmétique suivant : AB + CD = DC, dans lequel A, B, C et D représentent des chiffres *tous distincts* et *non nuls*. Ni itération, ni récursion ! Il s'agit simplement d'exprimer les contraintes sur les quatre variables et sur la retenue r de l'addition B+D, et de laisser `amb` faire son travail de backtrack…

```
> (solve-puzzle)
Solution : 18 + 24 = 42          ; faites afficher toutes les solutions…
```

Exercice 14.8 Ecrivez la fonction `(member x L)` avec une boucle, mais en exprimant la boucle comme un GOTO, en capturant le point d'entrée dans la boucle par une continuation, à l'instar de ce qui a été fait pour la factorielle dans le cours 14 page 25.

Exercice 14.9 On se propose d'implémenter un cas simple de **BREAK**. Poser un point d'arrêt [***breakpoint***] dans un calcul signifie arrêter le calcul pour faire diverses choses, puis être capable de le reprendre exactement là où il s'était arrêté. Dans l'exemple qui suit, on stoppe le calcul lorsque `n=2`, et l'on entre dans une boucle REP usuelle. En tapant `(resume v)`, on reprend le calcul en renvoyant la valeur `v` comme valeur de retour du `break` [en principe `v` est la valeur envoyée par le `break`, mais ce pourrait être une autre, pourquoi pas ?] :

```
(define (fac n)
  (if (= n 0)
      1
      (* (if (= n 2) (break n) n)      ; breakpoint si n=2
         (fac (- n 1)))))
```

```
> (fac 5)
Entering break. Call (resume v) to exit break loop with value v !
2
> (+ 2 3)              ; je fais [semblant de faire] divers calculs au toplevel…
5
> (resume 2)           ; puis je sors du break et revient dans le calcul récursif
120
>
```

Ah oui, quelle est la question ? Mais vous aviez deviné, n'est-ce pas ? Implémenter la fonction `(break x)` bien entendu…
Au fait, peut-on invoquer `resume` plusieurs fois, pour essayer diverses valeurs ?…

Exercice 14.10 **a)** En utilisant la fonction `(an-integer-between a b)`, programmez une fonction `(pythagoriciens n)` retournant de manière *non-déterministe* un triplet `(a b c)` d'entiers tels que $1 \leq a \leq b \leq c \leq n$ et $a^2 + b^2 = c^2$. Il y a 6 triplets pythagoriciens pour `n=20`, lesquels ?…

b) L'implémentation proposée de `(amb)` se trouve sur le site Web de Dorai Sitaram [*Teach Yourself Scheme in Fixnum Days §14.4.2*]. Vous êtes invités à y étudier l'élégante solution au problème du *coloriage avec 4 couleurs* d'une carte géographique, problème algorithmique très célèbre résolu en 1976 par Appel et Haken et une partie de la preuve a nécessité des calculs intensifs sur ordinateur. La solution développée en SCHEME avec `(amb)` est quasiment celle que l'on écrirait en PROLOG.