Université Montpellier-II
UFR des Sciences - Département Informatique - Licence Informatique
Programmation Applicative et Récursive

Cours No 10 : Fonctions récursives et Interprétation des programmes

Notes de cours

17 Récursivité et Interprétation des programmes

Réalisons un langage de programmation défini par sa fonction d'interprétation de ses expressions. Dotons le minimalement pour débuter.

- — Des types de base et leurs fonctions primitives : entiers, booléens
  - — des identificateurs, définis par "let"
  - — Une conditionnelle, nommée "si", ayant la même sémantique que la conditionnelle usuelle.
- — Une syntaxe. Pour simplifier, prenons celle de Scheme. Nous disposons d'un analyseur syntaxique prêt à l'emploi, implanté par la fonction "read".
- — Une sémantique des constructions du langage.
  Prenons celle de Scheme et mettons la en oeuvre via une fonction d'interprétation : `eval302`.
  Rendons la fonction d'interprétation utilisable via une boucle "toplevel" implantée par la fonction `scheme302`.

```
1  (define (scheme304)
2
3    ;; sans fonction utilisateur
4    ;; sans environnement, hors global
5    ;; sans gestion de la pile
6
7    (let ((**global-env (list (list '+ +) (list '- -) (list '* *) (list '/ /) (list '= =)
8                              (list 'car car) (list 'cdr cdr) (list 'cons cons) (list 'append
                                   append) (list 'list list)))
9          (**primitives '(+ - * / =))
10         (**returnToToplevel #f)
11         (**fin? #f))
12
13     (define (primitive? f) (memq f **primitives))
14
15     ;; Gestion des environnements
16     (define (env-value symbol env)
17       (let ((liaison (assq symbol env)))
18         (if liaison
19             (cadr liaison)
20             (error304 "variable_indéfinie" symbol))))
21
22     (define (eval304 e env)
```

```
23      (cond ((or (number? e) (boolean? e) (string? e) (procedure? e)) e)
24            ((symbol? e) (env-value e env))
25            ((list? e)
26             ;;les formes spéciales du langage
27             (case (car e)
28               ((fin) (set! **fin? #t) 'Au-revoir)
29               ((si) (eval-si e env))
30               ((let) (eval-let e env))
31               ((definir) (eval-definir e env))
32               (else
33                (cond
34                  ((primitive? (car e))
35                   ;; c'est une fonction primitive de notre langage, on passe la main
36                   ;; à notre interpréteur hôte pour appliquer la fonction, en prenant soin
37                   ;; d'évaluer préalablement les arguments
38                   (apply (eval304 (car e) env) (evlis (cdr e) env)))
39                  (#t (error304 "fonction inconnue" (car e)))))))
40            (#t (error304 "construction_inconnue" e))))

42    (define (eval-cite e env) (cadr e))

44    (define (eval-si e env)
45      (if (eval304 (cadr e) env)
46          (eval304 (caddr e) env)
47          (eval304 (cadddr e) env)))

49    (define (eval-let e env)
50      (let ((newenv (append (parallel-eval-bindings (cadr e) env) env)))
51        (eval-sequence (cddr e) newenv)))

53    (define (parallel-eval-bindings bindings env)
54      ;; rend un environnement augmenté des nouvelles liaisons des variables au
55      ;; résultat de l'évaluation des expressions correspondantes, dans l'environnement
56      ;; env
57      (if (null? bindings)
58          ()
59          (append (list (list (caar bindings) (eval304 (cadar bindings) env)))
60                  (parallel-eval-bindings (cdr bindings) env))))

62    (define (eval-sequence lexp env)
63      ;; evalue les expressions en séquence et rend
64      ;; la valeur de la dernière
65      (letrec ((boucle (lambda (lexp value)
66                         (if (null? lexp)
67                             value
68                             (boucle (cdr lexp) (eval304 (car lexp) env))))))
69        (boucle lexp ())))

71    (define (evlis l env)
72      ;; reçoit une liste d'expressions et rend la liste de leurs valeurs
```

```
73        (map (lambda (e) (eval304 e env)) l))
74
75      (define (eval-definir e env)
76        ;; permet d'ajouter une liaison
77        3
78      )
79
80      (define (error304 s l)
81        ; gestion primitive des exceptions
82        ; display + retour au toplevel
83        (display "Erreur: ") (display s) (display ", ") (display l) (display ".\n")
84        (**returnToToplevel **returnToToplevel))
85
86      (define (print304 e)
87        (display "= ") (display e) (display "\n"))
88
89      ;; Corps du let principal
90
91      (do ()
92        ;; la boucle while(true) selon Scheme
93
94        ;; pour quitter la boucle toplevel, écrire "(fin)"
95        ;; si la variable **fin vaut #t, sortie du toplevel
96        (**fin? 'A_bientôt) ;;
97
98        ;; gestion de base des exceptions.
99
100       ;; capture du point courant (une seule fois) pour y revenir après une exception
101       ;;
102       (or **returnToToplevel
103           (begin
104            (set! **returnToToplevel
105                  (call-with-current-continuation
106                   (lambda (k) k)))
107            (display "... Extra-ball\n")))
108
109       (print304
110        (eval304
111         (read)
112         **global-env))
113       )
114     )
115   )
116
117 (scheme304)
```