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)))
20        (if liaison
21            (cadr liaison)
22            (error304 "variable_indéfinie" symbol))))

24    (define (eval304 e env)
25      (cond ((or (number? e) (boolean? e) (string? e) (procedure? e))
            e)
26            ((symbol? e) (env-value e env))
27            ((list? e)
28             ;;les formes spéciales du langage
29             (case (car e)
30               ((fin) (set! **fin? #t) 'Au-revoir)
31               ((si) (eval-si e env))
32               ((let) (eval-let e env))
33               ((definir) (eval-definir e env))
34               (else
35                (cond
36                  ((primitive? (car e))
```

```
37                      ;; c'est une fonction primitive de notre langage, on
                            passe la main
40                      ;; à notre interpréteur hôte pour appliquer la fonction,
                            en prenant soin
41                      ;; d'évaluer préalablement les arguments
42                      (apply (eval304 (car e) env) (evlis (cdr e) env)))
43                     (#t (error304 "fonction inconnue" (car e)))))))
44               (#t (error304 "construction_inconnue" e))))

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

48     (define (eval-si e env)
49       (if (eval304 (cadr e) env)
50           (eval304 (caddr e) env)
51           (eval304 (cadddr e) env)))

53     (define (eval-let e env)
54       (let ((newenv (append (parallel-eval-bindings (cadr e) env)
             env)))
```

```
55          (eval-sequence (cddr e) newenv)))
58
59      (define (parallel-eval-bindings bindings env)
60        ;; rend un environnement augmenté des nouvelles liaisons des
              variables au
61        ;; résultat de l'évaluation des expressions correspondantes, dans
              l'environnement
62        ;; env
63        (if (null? bindings)
64            ()
65            (append (list (list (caar bindings) (eval304 (cadar
                  bindings) env)))
66                    (parallel-eval-bindings (cdr bindings) env))))
68      (define (eval-sequence lexp env)
69        ;; evalue les expressions en séquence et rend
70        ;; la valeur de la dernière
71        (letrec ((boucle (lambda (lexp value)
72                           (if (null? lexp)
```

```
73                          value
76                          (boucle (cdr lexp) (eval304 (car lexp)
                                env))))))
77              (boucle lexp ())))

79      (define (evlis l env)
80        ;; reçoit une liste d'expressions et rend la liste de leurs valeurs
81        (map (lambda (e) (eval304 e env)) l))

83      (define (eval-definir e env)
84        ;; permet d'ajouter une liaison
85        3
86      )

88      (define (error304 s l)
89        ; gestion primitive des exceptions
90        ; display + retour au toplevel
91        (display "Erreur: ") (display s) (display ", ") (display l)
              (display ".\n")
```

```
92        (**returnToToplevel **returnToToplevel))
95
96      (define (print304 e)
97        (display "= ") (display e) (display "\n"))

99      ;; Corps du let principal

101     (do ()
102       ;; la boucle while(true) selon Scheme

104       ;; pour quitter la boucle toplevel, écrire "(fin)"
105       ;; si la variable **fin vaut #t, sortie du toplevel
106       (**fin? 'A_bientôt) ;;

108       ;; gestion de base des exceptions.

110       ;; capture du point courant (une seule fois) pour y revenir après une
              exception
111       ;;
```

```
112       (or **returnToToplevel
115           (begin
116            (set! **returnToToplevel
117                  (call-with-current-continuation
118                   (lambda (k) k)))
119            (display "... Extra-ball\n")))

121       (print304
122        (eval304
123         (read)
124         **global-env))
125       )
126     )
127   )

129 (scheme304)
```