Programmation Fonctionnelle Avancée 2007-2008
L3-Info - Faculté des Sciences de Nice
http://deptinfo.unice.fr/~roy

Cours n°14
(euh, presque…)

Ouvrir la Porte du Contrôle

l'art subtil des continuations

Le contrôle ? Quel contrôle ?

- Une évaluation s'opère au sein d'un environnement :

```
($eval expr env)        ; cf cours 11 (11a et 11b)
```

- En un point donné de l'évaluation, `$eval` ne peut pas **contrôler** ce qui se passe. Il ne connaît pas le CONTEXTE du calcul (son histoire).
- Lui donner accès à ce contexte, c'est lui ouvrir les Portes du Contrôle.
- Nous allons entr'ouvrir trois portes :

① les exceptions
② les continuations
③ le non-déterminisme

Attraper une EXCEPTION

- Il s'est passé un phénomène exceptionnel. Scheme vient de **lever une exception** : l'exception `exn:fail:read`.

```
 > (read)
)
read: unexpected ')'
```

- Le traitement normal consiste à afficher le message de l'exception `(exn-message exn)`, puis à abandonner [comment ?] le contexte de calcul courant et à revenir à la boucle toplevel.

par une continuation, wait and see…

- Il y a beaucoup d'exceptions prédéfinies [Racket Reference §10.2.5]

```
exn:fail:read                  exn:fail:contract:variable
exn:fail:contract:arity        exn:fail:contract:divide-by-zero
exn:fail:read:eof              exn:fail
```

etc.

- On peut soi-même attraper l'exception au vol en positionnant son propre **handler d'exceptions** avec `with-handlers` :

```
(define (protected-read)
  (with-handlers ((exn:fail:read? (lambda (exn) 'ovni)))
    (printf "Entrez une expression ==> ")
    (read)))


> (let ((expr (protected-read)))
    (printf "Je viens de lire : ~a\n" expr))
Entrez une expression ==> foo)
Je viens de lire : ovni
```

sans erreur !

- On peut positionner plusieurs handlers :

```
(with-handlers ((p1? proc1) (p2? proc2) ...)
  ...)
```

où chaque `pi?` est de la forme `(lambda (exn) ...)`

• Le prédicat exn:fail? permet d'attraper n'importe quelle erreur, en particulier déclenchée par un appel à error :

```
(define (foo)
  (error "i am losing"))      ; lève l'exception exn:fail:user

> (with-handlers
      ([exn:fail? (lambda (exn)
                    (printf "BUG:~a\n" (exn-message exn))
                    1000)])
    (+ 1 (foo) 2))
BUG:i am losing
1000
```

• On peut aussi lever (*raise*) ses propres exceptions, mais nous n'en parlerons pas…


```
public class Bar {
  private static int foo() {
    return 1000/0;
  }

  public static void main(String[] args) {
    try {
      System.out.println(1 + foo() + 2);
    } catch (Exception e) {
      System.out.println("BUG:" + e.getMessage());
      System.out.println(2000);
    }
  }
}
```

```
BUG:/ by zero
2000
```


Les continuations

• Difficulté de **s'échapper d'un calcul** à cause de la pile de récursion [*control stack*] qui contient des calculs en attente qu'il faudrait abandonner…

```
> (+ (* 1000 (abort (* 2 3))) (+ 2000 3000))
6
```

bye-bye !…

• Comment définir la fonction abort ? Deux solutions :

soft

- transformer le programme en **style CPS** en traînant à tout moment la continuation courante du calcul.

hard

- utiliser les **vraies continuations** qui sont en Scheme des objets de 1ère classe.

Les continuations « soft »

Voir le cours 11 "Scheme en Scheme"

• La transformation CPS (*Continuation Passing Style*) introduit un nouveau paramètre fonctionnel : la continuation courante du calcul.

```
(define (k-foo x cont)        ; retourne (cont (foo x))
  ...)
```

• L'intérêt de ce paramètre « continuation » [une fermeture !] est :

i) de connaître à tout instant le futur du calcul.

ii) de pouvoir l'abandonner et glisser vers un autre futur.

iii) d'être un objet « de 1ère classe » que l'on peut stocker dans une structure de donnée ou passer en paramètre à une autre fonction.

• N.B. Ce style CPS généralise le style classique (*direct style*) pour lequel cont est l'identité (lambda (x) x).

• Reprenons un évaluateur d'expressions arithmétiques (cours 11.1) :

```
(define ($eval expr)
  (if (number? expr)
      expr
      (let ((Lvals ($evlis (cdr expr))))        ; les arguments
        (case (car expr)
          ((+) (apply + Lvals))
          ((-) (apply - Lvals))
          . . . . . .
          (else (error "Unknown op" (car expr)))))))


(define ($evlis Lexpr)
  (if (null? Lexpr)
      '()
      (cons ($eval (car Lexpr)) ($evlis (cdr Lexpr)))))
```

```
> ($eval '(+ (* 2 3 4) 5))
29
```



```
(define (k-$eval expr cont)
  (if (number? expr)
      (cont expr)
      (k-$evlis (cdr expr)
        (λ (Lvals)
          (case (car expr)
            ((+) (cont (apply + Lvals)))
            ..........
            (else (error "Unknown op" (car expr))))))))

(define (k-$evlis Lexpr cont)
  (if (null? Lexpr)
      (cont'())
      (k-$eval (car Lexpr)
        (λ (v) (k-$evlis (cdr Lexpr)
                 (λ (L) (cont (cons v L))))))))
```

mise en CPS !

```
> (k-$eval '(+ (* 2 3 4) 5) (λ (x) (* x 1000)))
```


• A tout moment je tiens la continuation du calcul dans la variable cont. Supposons que je souhaite **abandonner la continuation courante** pour m'échapper du calcul, pour implémenter un `return` comme en C ou Java :

```
(define (k-$eval expr cont)
  (cond ((number? expr) (cont expr))
        ((eq? (car expr) 'return) (k-$eval (cadr expr) identity))
        (else (k-$evlis (cdr expr)
                (λ (Lvals)
                  (case (car expr)
                    ((+) (cont (apply + Lvals)))
                    ..................
                    (else (error "Unknown op" (car expr))))))))))
```

abandon de cont

```
> (k-$eval '(+ (* 2 3 4) 5) id)
29
> (k-$eval '(+ (* 2 (return (* 10 10)) 4) 5) (λ (x) (* x 1000)))
```


• Ex : **capture de la continuation courante** sur le même exemple :

```
(define the-cont '?)        ; une continuation globale

(define (k-$eval expr cont)
  (cond ((number? expr) (cont expr))
        ((eq? (car expr) 'capture!) (set! the-cont cont))
        (else (k-$evlis (cdr expr)
                (λ (Lvals)
                  (case (car expr)
                    ((+) (cont (apply + Lvals)))
                    ..................
                    (else (error "Unknown" (car expr))))))))))
```

capture [et abandon]

```
> (k-$eval '(+ (* 2 (capture!) 4) 5) (λ (x) (* x 1000)))
> the-cont
#<procedure>
> (the-cont 1)
```

*Capture de **cont**exte !!!*

• Mais les **continuations à plusieurs variables** ont aussi leur utilité, par exemple pour un calcul des nombres de Fibonacci avec un nombre linéaire d'opérations :

une continuation à 2 variables…

```
(define (k-fib n cont)          ; → (cont (fib n) (fib (+ n 1)))
  (if (= n 0)
      (cont 0 1)
      (k-fib (- n 1) (λ (a b) (cont b (+ a b))))))

> (time (k-fib 200 (λ (a b) a)))        ; calcul de fib(200)
[Time ≈ 0 sec]
280571172992510140037611932413038677189525

> (map (λ (n) (k-fib n (λ (a b) a))) (iota 15 0))
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
```


• Ex: **application aux générateurs**. Soit à résoudre un problème ayant plusieurs solutions [peut-être une infinité !]. Etre capable de calculer un couple <sol,cont> où sol est la première solution disponible et cont la continuation de la recherche du prochaine couple <sol, cont>.

• Générateur des éléments d'une liste vérifiant un prédicat donné :

```
(define (solution p? L)
  (cond ((null? L) '*fail*)
        ((p? (car L)) (cons (car L)
                            (λ () (solution p? (cdr L)))))
        (else (solution p? (cdr L)))))
```

*a « **thunk** » un « glaçon »*

Plutôt que relancer récursivement la recherche, on « gèle » le calcul dans une lambda pour pouvoir le relancer plus tard !


```
(define (suivante sol)
  (if (eq? sol '*fail*)
      '*fail*
      ((cdr sol))))
```

```
(define (courante sol)
  (if (eq? sol '*fail*)
      '*fail*
      (car sol)))
```

```
> (define s (solution integer? '(1 et 2 font 3)))
> s
(1 . #<procedure>)
> (courante s)
1
> (set! s (suivante s))
> (courante s)
2
> (set! s (suivante s))
> (courante s)
3
> (set! s (suivante s))
> (courante s)
*fail*
```

*La solution courante + une **promesse** de calcul…*

• Un **ARPENTEUR GÉNÉRAL** en profondeur préfixe [parcours exhaustif ou pas !] dans un arbre binaire d'expression :

```
(define (visiter A k-noeud k-feuille cont)
  (if (feuille? A)
      (k-feuille A cont)
      (k-noeud A
               (lambda () (visiter (fg A)
                                   k-noeud
                                   k-feuille
                                   (lambda ()
                                     (visiter (fd A)
                                              k-noeud
                                              k-feuille
                                              cont)))))))
```

(k-noeud noeud cont) où cont est un glaçon

(k-feuille feuille cont) où cont est un glaçon

(cont) pour dégeler un glaçon

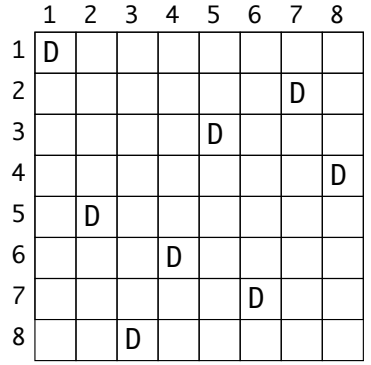
- Exemple 1 : la liste des feuilles

```
> (visiter '(+ (* -2 (/ 3 x)) (/ 4 -5))
           (λ (noeud cont) (cont))
           (λ (feuille cont) (cons feuille (cont)))
           (λ () '()))
(-2 3 x 4 -5)
```

- Exemple 2 : le premier sous-arbre de racine /

```
> (visiter '(+ (* -2 (/ 3 x)) (/ 4 -5))
           (λ (noeud cont)
             (if (equal? (racine noeud) '/) noeud (cont)))
           (λ (feuille cont) (cont))
           (λ () '*fail*))
(/ 3 x)
```


Ex : **application au backtrack**. Le problème des N dames sur un échiquier. Placer les N dames de sorte qu'il n'y ait aucun couple de dames en prise.




```
((8 4) (7 2) (6 7) (5 3) (4 6) (3 8) (2 5) (1 1))
```

colonne ligne

- J'ai à résoudre un problème de N dames. J'ai déjà placé les k premières dames dans la position L et j'essaye de placer la (k+1)-ème dame en ligne i. Pour backtracker, j'invoque (backtrack).

```
(define (queens N k L i backtrack)
  (cond ((= k N) L)
        ((> i N) (backtrack))
        ((ok? (+ k 1) i L)
         (queens N (+ k 1)
                 (cons (list (+ k 1) i) L) 1
                 (λ () (queens N k L (+ i 1) backtrack))))
        (else (queens N k L (+ i 1) backtrack))))


> (queens 3 0 '() 1 (λ () 'no-more))
no-more
> (queens 8 0 '() 1 (λ () 'no-more))
((8 4) (7 2) (6 7) (5 3) (4 6) (3 8) (2 5) (1 1))
```


Les continuations « hard »

- Scheme procure des objets de type « continuation » avec une primitive call/cc [*call-with-current-continuation*] permettant de capturer la continuation courante d'un calcul [pour y revenir rapidement par exemple] :

capture

```
> (+ 3 (call/cc (lambda (k) (* 3 4))) 5)
20
> (+ 3 (call/cc (lambda (k) (* (/ 5 (k 8)) 4))) 5)
16
> (+ 3 (call/cc (lambda (k) (set! the-cont k) 4)) 5)
12
> (the-cont 1)
9
```

photographie de la continuation courante !

Procedure: (call/cc proc)

R5RS

proc *must be a procedure of one argument. The procedure* call/cc *packages up the current continuation as an « escape procedure » and passes it as an argument to proc. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created [...].*

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

• Application : **s'échapper d'un calcul récursif** en abandonnant les calculs en attente sur la pile de récursion :

```
(define (prod L)
  (call/cc (lambda (k)
    (letrec ((recur
              (lambda (L)
                (cond ((null? L) 1)
                      ((zero? (car L)) (k 0))  ; abandon !
                      (else (mul (car L) (recur (cdr L))))))))
      (recur L)))))
```

```
(define (mul a b)
   (set! cpt (add1 cpt))
   (* a b))
```

```
> (set! cpt 0)
> (printf "prod=~a cpt=~a~n" (prod '(2 3 4 5 6)) cpt)
prod=720 cpt=5
> (set! cpt 0)
> (printf "prod=~a cpt=~a~n" (prod '(2 3 4 0 5 6)) cpt)
prod=0 cpt=0
```

5 multiplications effectuées

Aucune multiplication effectuée !!!

• L'exemple du « abort » de la page 7 qui s'échappait d'un calcul profond et retournait directement un résultat au toplevel en ignorant les calculs en attente est obtenu en **capturant la continuation du toplevel** :

```
> (define abort '?)
> (call/cc (lambda (k) (set! abort k)))     ; capture !
> abort
#<continuation>
> (+ (* 3 (abort (* 4 5))) (/ 6 7))
20
```

• Une définition approchée de la fonction error pourrait alors être :

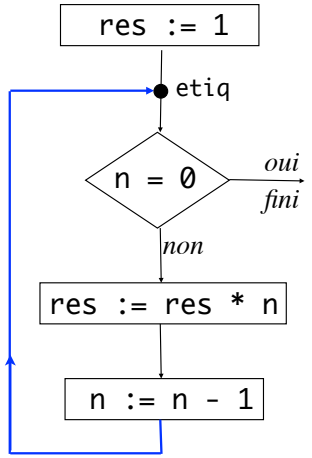
```
(define (error msg)
  (abort (printf "~a~n" msg)))
```


• Mais l'inévitable factorielle, passant par là, entendit du bruit et entra :

```
  res := 1
etiq
  IF n = 0 THEN RETURN res
  res := res * n
  n := n - 1
  GOTO etiq
```

Comment modéliser un GOTO en Scheme ?

ou : qu'est-ce qu'une étiquette ?...



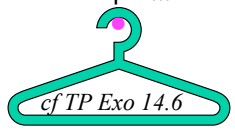
```
(define (fac n)
  (call/cc
    (λ (return)
      (let ((GOTO '?) (res 1))
        (call/cc (λ (cont) (set! GOTO cont)))     ; capture !
        ; etiquette
        (if (zero? n)
            (return res)
            (begin (set! res (* res n))
                   (set! n (- n 1))
                   (GOTO 'etiquette))))))))
```

```
> (fac 10)
3628800
```

Mmmm...

```
int fac(int n) {
  int res = 1;
  etiq:
  if(n == 0) return res;
  res = res * n;
  n = n - 1;
  goto etiq;
}
```


- On peut donc penser une boucle comme une **reprise de continuation**. Mézalor, pourquoi ne pas en faire une **macro** générale loop ?...




```
(define-syntax loop
  (syntax-rules ()
    ((loop e1 e2 ...)
     (call/cc (λ (exitloop)
                (letrec ((iter (λ () e1 e2 ... (iter))))
                  (iter)))))))

(define (afficher L)        ; affichage d'une liste à la Python...
  (printf "[")
  (if (not (null? L))
      (loop (write (car L))
            (if (null? (cdr L)) (exitloop '?))
            (display ",")
            (set! L (cdr L))))
  (printf "]"))
```

point de sortie au milieu !

```
> (afficher (range 1 11))
[1,2,3,4,5,6,7,8,9,10]
```

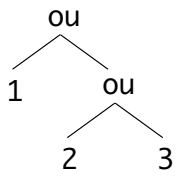

Application au non déterminisme

- Modéliser un style de recherche en profondeur « à la Prolog » avec **gestion automatique du backtrack**. Par exemple avec **l'opérateur de choix ambigu** amb de McCarthy (1962).

- `(amb e1 e2 ...)` retourne de manière non déterministe la valeur de l'un des e_i. Par exemple `(amb 1 2 3)` peut retourner 1, 2 ou 3.

- `(amb)` échoue et essayera de reprendre le calcul au dernier point de choix.

```
> (amb 1 2 3)
1
> (amb)
2
> (amb)
3
> (amb)
ERROR : amb tree exhausted
```




```
> (amb 1 (amb))
1
> (amb)
amb tree exhausted
```

`(initialize-amb-fail)`

```
> (if (amb #t #f) (amb) 1)
1
> (amb)
amb tree exhausted
```

```
> (amb (amb) 1)
1
> (amb)
amb tree exhausted
```

```
> (list (amb 1 2) (amb 'a 'b))
(1 a)
> (amb)
(1 b)
> (amb)
(2 a)
> (amb)
(2 b)
> (amb)
amb tree exhausted
```

```
> (if (amb #t #f) 1 2)
1
> (amb)
2
> (amb)
amb tree exhausted
```

• On peut bien entendu s'en servir pour programmer des **fonctions non-déterministes** :

```
(define (an-integer-between a b)        ; un entier de [a,b]
  (if (> a b)
      (amb)
      (amb a (an-integer-between (+ a 1) b))))


> (an-integer-between 2 5)
2
> (amb)
3
> (amb)
4
> (amb)
5
> (amb)
amb tree exhausted
```


• On peut aussi vouloir **positionner des contraintes** avec ASSERT pour restreindre l'espace des solutions :

```
(define (assert bool)        ; je force bool à être vrai
  (if (not bool) (amb)))     ; sinon, backtrack !


> (let ((i (an-integer-between 10 20)))
    (assert (premier? i))
    i)
11
> (amb)
13
> (amb)                                  ? Joli?
17                                       #t
> (amb)
19
> (amb)
amb tree exhausted
```


Le code de amb (difficile)

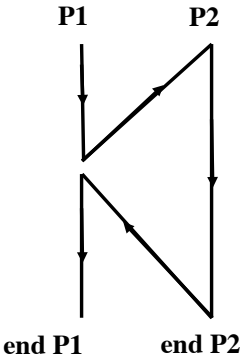
```
(define amb-fail '?)

(define (initialize-amb-fail)
  (set! amb-fail (lambda () (error "amb tree exhausted"))))

(initialize-amb-fail)

(define-syntax amb
  (syntax-rules ()
    ((amb e ...)
     (let ((+prev-amb-fail amb-fail))
       (call/cc
         (lambda (+sk)
           (call/cc
             (lambda (+fk)
               (set! amb-fail
                     (lambda ()
                       (set! amb-fail +prev-amb-fail)
                       (+fk 'fail)))
               (+sk e)))
           ...
           (+prev-amb-fail)))))))
```

+sk = *success cont.*
+fk = *failure cont.*

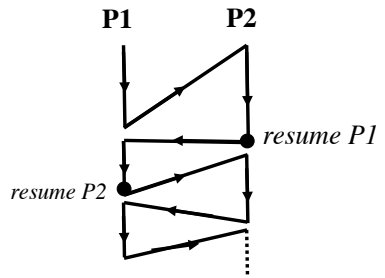
• L'opérateur amb est disponible en Racket avec #lang sicp.

• Il existe bien d'autres applications des continuations « hard » pour modéliser des structures de contrôle inexistantes à priori en Scheme. Par exemple les **coroutines**...

P1 appelle P2



coroutinage entre P1 et P2



• Scheme (et notamment Racket) est un laboratoire de langages !