12. Abstraction procédurale

Principe. La notion de procédure est la clef de la décomposition d'un problème en sous-problèmes. Le fait qu'en Scheme une procédure puisse accepter des procédures comme données et produire des procédures comme résultats rend le langage spécialement adapté à *l'abstraction procédurale*.

Conséquence. En programmation comme en mathématique, il est souvent opportun de reconnaître en un problème donné un cas particulier d'un problème plus général, et même de chercher d'emblée à résoudre le problème général, ce qui produira une procédure largement réutilisable dans des contextes variés.

Application. On va voir comment une procédure itérative de calcul de la racine carrée peut se généraliser en une procédure très générale de mise en œuvre d'un processus d'approximation.

Méthode itérative de calcul de $\sqrt{x}$

$y_0 = 1 \qquad y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$

Si $x = 2$: $1\,,\ \frac{1}{2}\left(1 + \frac{2}{1}\right) = 1.5\,,\ \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.4167\,,\ \ldots$

```
(define sqrt-iter
  (lambda (guess x)
    (display "  ") (write guess)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x))))

(define improve
  (lambda (guess x) (/ (+ guess (/ x guess)) 2)))

(define good-enough?
  (lambda (guess x) (< (abs (- (square guess) x)) 0.0000000001)))

(sqrt-iter 1.0 2.0)
1.  1.5  1.4166666666666665  1.4142156862745097  1.4142135623746899
;Value: 1.4142135623746899
```

Abstraction et généralisation I

Généralisation élémentaire : passer de la racine carrée à la racine $p$ième.

Méthode itérative de calcul de $\sqrt[p]{x}$.

$$y_0 = 1 \qquad\qquad y_{n+1} = \frac{1}{p}\left((p-1)y_n + \frac{x}{y_n^{p-1}}\right)$$

```
(define sqrt-p-iter
  (lambda (guess x p) ...

(define improve
  (lambda (guess x p)
    (/ (+ (* (- p 1) guess)
          (/ x (expt guess (- p 1)))) p)))

(define good-enough?
  (lambda (guess x p)
    (< (abs (- (expt guess p) x)) 0.00000001)))

(sqrt-p-iter 1.0 729.0 3)   9.
(sqrt-p-iter 1.0 2.0 2)     1.4142135623746899
```

Abstraction et généralisation II

Généralisation moins élémentaire :
passer de l'équation $y^p - x = 0$ à l'équation $f(y) = 0$.

$y_0 = 1 \qquad y_{n+1} = y_n - \frac{f(y_n)}{Df(y_n)}$

```
(define solve
  (lambda (guess f Df)
    (if (good-enough? guess f)
        guess
        (solve (improve guess f Df) f Df))))

(define improve
  (lambda (guess f Df) (- guess (/ (f guess) (Df guess)))))

(define good-enough?
  (lambda (guess f) (< (abs (f guess)) 0.1)))

(solve 1.0 (lambda (y) (- (expt y 3) 729.0))
           (lambda (y) (* 3 (expt y 2))))      9.0000220253469
```

Abstraction et généralisation III

Résolution itérative de $f(y) = 0$ avec calcul approximatif de la dérivée

```
(define newton
  (lambda (gu f dx)
    (if (good-enough? gu f)
        gu
        (newton (improve gu f dx) f dx))))

(define deriv
  (lambda (f dx) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))

(define improve
  (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))

(define good-enough?
  (lambda (gu f) (< (abs (f gu)) 0.1)))

(newton 1.0 (lambda (y) (- (expt y 3) 729.0)) 0.0001)  9.000022153425999
(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)         0.7503675298583334
(cos 0.7503675298583334)                               0.731438296864949
```

(Ici, `good-enough?` ... ne mérite pas son nom.)

Abstraction et généralisation IV

```
(define newton
  (lambda (gu f dx)
    (let* ((deriv
             (lambda (f dx)
               (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
           (improve
             (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))
           (good-enough?
             (lambda (gu f) (< (abs (f gu)) 0.001))))
      (if (good-enough? gu f)
          gu
          (newton (improve gu f dx) f dx)))))
```

```
(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)       0.7391131535431725
(cos 0.7391131535431725)                             0.7390662580950105
```

(`good-enough?` a été modifié.)

Abstraction et généralisation V

Calcul itératif de point fixe : résoudre $x = f(x)$

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
             (lambda (gu f) (< (abs (- gu (f gu))) 0.001))))
      (if (good-enough? gu f) gu (fixpoint (f gu) f)))))

(fixpoint 1.0 cos)                         0.7395672022122561
(cos 0.7395672022122561)                   0.7387603198742113
(fixpoint 1.0 (lambda (x) (/ 2 x)))        ...
```

Amélioration par lissage

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
             (lambda (gu f) (< (abs (- gu (f gu))) 0.001)))
            (improve
             (lambda (gu f) (/ (+ gu (f gu)) 2))))
      (if (good-enough? gu f) gu (fixpoint (improve gu f) f)))))

(fixpoint 1.0 (lambda (x) (/ 2 x)))        1.414...
```

Abstraction et généralisation VI

Idée : `fixpoint` et `newton` sont deux instances de `iterative-improve`. `iterative-improve` prend comme arguments des procédures `good-enough?` et `improve` et renvoie comme valeur une procédure `f` telle que `(f gu)` soit `(if (good-enough? gu) gu (f (improve gu)))`

On doit donc

- définir `iterative-improve`
- écrire `fixpoint` et `newton`
  comme instances de `iterative-improve`

Ceci permettra des appels tels que

```
((fixpoint cos) 1.0)                              0.7392146118880453
((newton (lambda (x) (- x (cos x))) 0.001) 1.0)  0.7391155232281558
```

Abstraction et généralisation VII

```
(define iterative-improve
  (lambda (good-enough? improve)
    (lambda (gu)
      (letrec
        ((f (lambda (g) (if (good-enough? g) g (f (improve g))))))
        (f gu)))))

(define fixpoint
  (lambda (f)
    (iterative-improve (lambda (gu) (< (abs (- gu (f gu))) 0.001))
                       (lambda (gu) (/ (+ gu (f gu)) 2)))))

(define newton
  (lambda (f dx)
    (let ((deriv
            (lambda (f) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))))
      (iterative-improve
        (lambda (gu) (< (abs (- gu (f gu))) 0.001))
        (lambda (gu) (- gu (/ (f gu) ((deriv f) gu))))))))
```

Itérateur I

On appelle $n$ième itérée de la fonction f de D dans D
la composée de n fonctions égales à f.

> *Ecrire une fonction* `iter`
> *tel que pour tout naturel* `n`*,*
> `(iter n)` *soit la fonction*
> *qui à toute fonction* `f` *auto-composable*
> *associe la* `n`*ième itérée de* `f`*.*

La solution est immédiate, mais il faut veiller à respecter le type fonctionnel des objets manipulés.

```
(define iter                                  ;; On définit iter
  (lambda (n)                                 ;; une fonction à un argument
    (lambda (f)                               ;; (iter n) associe à f
      (if (zero? n)                           ;; si n vaut 0
          (lambda (x) x)                      ;; la fonction identité
          (compose f ((iter (- n 1)) f))))))  ;; sinon la composée de ...
```

Itérateur II

```
(define iter      ;; variante
  (lambda (n)
    (lambda (f)
      (lambda (x)
        (if (zero? n) x (f (((iter (- n 1)) f) x)))))))

(define it        ;; scinder la difficulté
  (lambda (n f x)
    (if (= n 0) x (f (it (- n 1) f x)))))

(define iter-bis  ;; autre variante, équivalente
  (lambda (n)
    (lambda (f) (lambda (x) (it n f x)))))

(((iter     3) cos) 1)   .6542897904977791
(((iter-bis 3) cos) 1)   .6542897904977791
(cos (cos (cos 1)))      .6542897904977791
(it 3 cos 1)             .6542897904977791
```