OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

Cours n°9

http://deptinfo.unice.fr/~roy

Les Entrées-Sorties & le Web

pp. 295-310

Rudiments sur les E/S

• **Entrée** : donnée que l'on envoie dans un port d'entrée pour qu'elle y soit traitée automatiquement. → *input port :* clavier, fichier, Web, string,...

• **Sortie** : donnée produite à la suite d'un traitement, et envoyée sur un port de sortie. → *output port :* écran, fichier, imprimante, Web, string...

• Les **ports** en Scheme sont des objets de 1ère classe :

```
> (port? (current-output-port))
#t
```

http://www.scheme.com/tspl3/io.html

• Les données sont reçues d'un port ou envoyées vers un port sous la forme d'une suite de caractères.

• Nous prendrons dans ce cours plutôt l'optique d'Emacs qui traite des flots d'objets Lisp. L'unité d'entrée-sortie ne sera pas le caractère [trop bas niveau] mais l'expression Scheme :

nombre, symbole, booléen, caractère, string, vecteur, doublet, ...

• "Une chaîne de catactères" sera vue comme un seul objet Scheme (en fait la *représentation externe* d'un objet Scheme).

• Un fichier contenant une liste de 5000 entiers ne contiendra en réalité qu'un seul objet Scheme, lisible par ***une seule instruction de lecture*** !

• On pourra déposer dans le fichier un arbre binaire contenant 2000 éléments à l'aide d'***une seule instruction d'écriture*** !

Pourquoi faire compliqué ?

La lecture au clavier

• Une fonction générale **(read)** pour lire... un objet Scheme :

```
(define (doubler)
  (printf "Entrez un nombre ou une liste de nombres : ")
  (define x (read))
  (cond ((number? x) (* x 2))
        ((list? x) (map (lambda (e) (* e 2)) x))
        (else (error "Entrée incorrecte !" x))))
```

```
> (doubler)
Entrez un nombre ou une liste de nombres : (2 5 8)
(4 10 16)
>
```

une boîte d'E/S !

N.B. Une liste `(a b c)` est lisible, ainsi qu'une string "abc", un nombre `-128` ou un booléen `#t`, mais pas `#<procedure>` par exemple !

• Attention, **l'expression lue n'est pas évaluée** !

```
(define (valeur-en-0)
  (printf "Entrez une fonction f : ")
  (define f (read))
  (printf "Sa valeur en 0 est ~a\n" (f 0)))

> (valeur-en-0)
Entrez une fonction : (lambda (x) (+ x 8))
```
procedure application : expected procedure, given: (lambda (x) (+ x 8))

• Il faut donc parfois utiliser **(eval e ns)**, qui évalue *dans un espace de noms* !

```
(define ns (make-base-namespace))

(let ((f (eval (read) ns)))
  (f 0))

> (valeur-en-0)
Entrez une fonction : (lambda (x) (+ x 8))
Sa valeur en 0 est 8
```

Eviter eval si possible !

• La fonction **(read-line)** permet de lire une suite de caractères, et les assemble en une chaîne, sans la fin de ligne :

```
> (begin (printf "---> ") (read-line))
---> voici quelques caractères !
"voici quelques caractères !"
>
```

• La fonction **(read-from-string str)** permet de lire des expressions Scheme dans une chaîne de caractère `str` :

```
> (require mzlib/string)      ; en voie de dépréciation...
> (read-from-string "12 juin 1978")
12
> (read-from-string-all "12 juin 1978")
(12 juin 1978)
```

cf page 18

Bien pratique lorsqu'on a des objets Scheme dans une chaîne, par exemple dans le text-field d'une interface graphique...

N.B. L'inverse de `read-from-string` est `(expr->string expr)`.

Ecriture sur l'écran

• Trois procédures normalisées, sans résultat :

(write x) *; affiche la valeur de x sous une forme lisible par (read)*
(display x) *; affiche la valeur de x pour des humanoïdes*
(newline) *; va à la ligne :* `(display "\n")`

```
> (write "a short string")
"a short string"
> (display "a short string")
a short string
> (write #\a)
#\a
> (display #\a)
a
```

mais 3 objets Scheme pour read !

• Dans ce cours, nous utiliserons plutôt la procédure **printf** sans résultat, bien qu'elle ne soit pas vraiment normalisée en Scheme !

`(printf` *format-string* `v ...)`

• *format-string* est une **chaîne de formatage**, pouvant contenir des **jokers ~a** [*display*] et **~s** [*write*], ainsi que des sauts de ligne **\n**. A chaque joker est associé une expression dont la valeur remplacera le joker :

```
> (printf "La racine carrée de ~a est ~a\n" 2 (sqrt 2))
La racine carrée de 2 est 1.4142135623730951
> (printf "Hier j'ai vu \"Orange mécanique\" !\n")
Hier j'ai vu "Orange mécanique" !
```

• La fonction **format** est analogue à `printf` mais *retourne* la chaîne en résultat sans l'afficher :

```
> (format "La racine carrée de ~a est ~a" 2 (sqrt 2))
"La racine carrée de 2 est 1.4142135623730951"
```

Lecture dans un fichier-disque

• La plupart des fonctions précédentes ont un analogue fonctionannt sur un port p d'entrée ou de sortie. Exemple : **(read p)** permet de lire la première expression Scheme disponible dans le port d'entrée p.

• Comment associer un **port d'entrée** à un fichier `f` sur le disque ?

- soit en ouvrant et fermant **manuellement** les ports avec :

```
(open-input-file f)      ; String → Port
(close-input-port p)     ; Port → void
```

- soit **automatiquement** avec :

```
(call-with-input-file f
  (lambda (p-in)      ; p-in est un port associé à f
    ...))
```

• Nous privilégierons la seconde manière !

• Exemple : soit à afficher et additionner tous les nombres contenus dans un fichier `"foo.dat"`.

```
; fichier "foo.dat"

-22.5          14     -2
   45   -15    1/2

  33
100
```

• Divers types de nombres, rangement discrétionnaire...

```
(define (somme-fichier f)
  (call-with-input-file f
    (lambda (p-in)
      (define (iter nb somme)
        (define x (read p-in))
        (if (eof-object? x)
            (printf "~a nombres de somme ~a\n" nb somme)
            (iter (+ nb 1) (+ somme x))))
      (iter 0 0))))
```

```
> (somme-fichier "foo.dat")
8 nombres de somme 153.0
>
```

Lecture dans un fichier de définitions Scheme

• ATTENTION : on lit en réalité des *expressions* Scheme. Comptons le **nombre de définitions dans un fichier de fonctions Scheme** !

```
(define (nb-definitions f)          ; incorrect si f est un module !
  (call-with-input-file f
    (lambda (p-in)
      (define (iter nb)
        (define x (read p-in))      ; lecture d'une expression Scheme
        (cond ((eof-object? x) nb)
              ((and (list? x) (equal? (car x) 'define))
               (iter (+ nb 1)))
              (else (iter nb))))
      (iter 0))))
```

```
; myprog.rkt
(define …)
(define …)
(foo 5)
(define …)
```

```
> (nb-definitions "myprog.rkt")
3
```

Et si le fichier Scheme est un module ?

• Une difficulté technique lorsque le fichier myprog.rkt débute par la directive `#lang racket`. Il s'agit alors d'un **module**, dont la structure est en réalité `(module myprog racket (#%module-begin ...))`, ne contenant qu'une seule grosse liste, donc lisible par un seul `(read)` !!! La directive `#lang` n'est donc pas autre chose qu'une abréviation qui expanse le contenu du fichier en une grosse et unique liste.

• Et pour lire sans erreur sur le `#lang` : `(read-accept-reader #t)`

```
#lang racket
;;; myprog.rkt
(read-accept-reader #t)
(call-with-input-file "myprog.rkt"
  (lambda (p-in)
    (printf "~a\n" (read p-in))))
```

```
(module myprog ...)
```

Ecriture dans un fichier-disque

• Comment associer un **port de sortie** à un fichier `f` sur le disque ?

- soit en ouvrant et fermant **manuellement** les ports avec :

```
(open-output-file f)     ; String → Port
(close-output-port p)    ; Port →
```

- soit **automatiquement** avec :

```
(call-with-output-file f
  (lambda (p-out)        ; p-out est un port associé à f
    ...)
  [#:exists m])
```

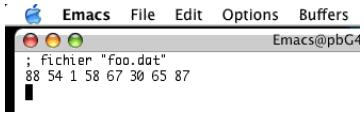
Le mode optionnel exprime ce qui se passe si le fichier existe déjà :
- 'replace [écrase le fichier s'il existe déjà]
- 'append [ajoute à la fin du fichier]
- 'error [par défaut , déclenche une erreur]

• Exemple : soit à créer un fichier contenant une ligne de commentaire avec son nom, puis une suite d'entiers aléatoires sur une même ligne. J'utilise fprintf ainsi que **la vraie boucle for** de Racket :

```
(define (creer-fichier-nombres f)
  (call-with-output-file f
    (lambda (p-out)
      (fprintf p-out "; fichier \"~a\"\n" f)
      (for ([i (in-range 8)])  ; à la Python, i ∈ [0,7]
        (fprintf p-out "~a " (random 100)))
      (fprintf p-out "\n"))
    #:exists 'replace))
```

```
> (creer-fichier-nombres "foo.dat")
> (file-exists? "foo.dat")
#t
```

N.B. `(fprintf p ...)` *pour écrire dans un port p.*



Déléguer du travail à l'OS !

• On ne programme pas des tâches que l'OS [Unix] peut faire tout seul !

```
(require racket/system)
```

• La fonction `(system cmd)` permet de passer à l'O.S. une chaîne `cmd` contenant une commande du shell exécutée de manière **synchrone** :

```
> (list (system-type 'os) (system-type 'word))
(macos-x 64)          ; Racket tourne en 64 bits sur une machine Unix
> (system "ls -l")
total 64
-rwxrwxrw-  1 jpr  admin  24064  20 Feb 16:14 TP9.doc
-rwxrwxrw-  1 jpr  admin   1565  19 Feb 14:28 code9.scm
drwxrwxrw-  1 jpr  admin   1024  20 Feb 15:26 cours9.key
-rwxrwxrw-  1 jpr  admin     45  20 Feb 15:27 foo.dat
-rwxrwxrw-  1 jpr  admin   3856  16 Feb 17:49 output-file.gif
#t
```

• On peut modifier la variable PATH avec **putenv** et **getenv**.

• Comment récupérer le résultat de la commande dans une chaîne ???

• Idée : rediriger la sortie courante vers un port associé à une chaîne, donc **considérer une chaîne comme un fichier** !

```
(define (get-output-cmd str)
  (define port (current-output-port))   ; on sauve le port courant
  (define p-out (open-output-string))   ; une chaîne en port de sortie
  (current-output-port p-out)           ; changer de port courant
  (system str)                          ; appel système !
  (close-output-port p-out)             ; fermer le port courant
  (current-output-port port)            ; restorer l'ancien port
  (get-output-string p-out))            ; récupérer la sortie
```

```
> (get-output-cmd "wc -l myprog.rkt")
"      6 myprog.rkt\n"
```

L'avantage de consulter la doc...

⇔

```
(with-output-to-string
  (lambda () (system "wc -l myprog.rkt")))
```

• Racket possède une importante API système. Il existe au moins deux autres moyens de récupérer le résultat d'une ligne de commande.

`(process <command>)`

• Analogue à `system`, mais exécutée de manière **asynchrone**. Elle retourne cinq résultats (voir la doc).

```
> (define res (process "ls -la"))  ; --> (ip op id iperr proc)
> ((fifth res) 'exit-code)         ; 'kill, 'wait, etc.
0                                  ; processus terminé OK
> (port->string (first res))
"total 64\n-rwxrwxrw  jpr admin  24064  20 Feb..."
```

; ne pas oublier de fermer les ports ip, op, iperr !

`(with-output-to-string (lambda () …))`

• On redirige la sortie vers un port chaîne :

```
> (with-output-to-string
    (lambda () (system "date")))
"Thu Nov 14 09:11:49 CET 2013\n"
```

SOLUTION 2 Se les programmer soi-même à partir de primitives de plus bas niveau :

```
(define (expr->string e)
  (with-output-to-string
    (lambda ()
      (printf "~s" e))))

(define (read-from-string-all str)
  (with-input-from-string str
    (lambda ()
      (define (eat-all)
        (define x (read))
        (if (eof-object? x)
            '()
            (cons x (eat-all))))
      (eat-all))))
```

```
> (expr->string '(+ 1 2))
"(+ 1 2)"
> (read-from-string-all "(+ 1 2) 3 4")
((+ 1 2) 3 4)
```

Ex: read-from-string et read-from-string-all sont **dépréciées** !

• Oui, un langage de programmation évolue, et **déprécie** parfois certaines primitives… Il est donc découragé de les utiliser. Alors ?…

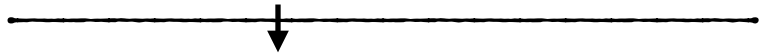
SOLUTION 1 Chercher si le langage ne propose pas une autre (nouvelle) manière. Par exemple pour `expr->string` :

```
> (require mzlib/string)
> (expr->string '(+ 1 2))
"(+ 1 2)"
```

```
> (require racket/format)
> (~s '(+ 1 2))
"(+ 1 2)"
```

• Ou encore pour `read-from-string-all` :

```
> (require mzlib/string)
> (read-from-string-all "1 (+ 2 3) 4")
(1 (+ 2 3) 4)
```



```
> (port->list read (open-input-string "1 (+ 2 3) 4"))
(1 (+ 2 3) 4)
```

Programmer un client Web

• But : automatiser une tâche sur le Web. Par exemple programmer un **client** qui va récupérer automatiquement une page HTML ou une image GIF sur un **serveur** pour la traiter ensuite.

• Connectons-nous sur un serveur de cinéma :

```
(define-values (p-in p-out)
  (tcp-connect "www.boncinema.fr" 80))
```

• La fonction **tcp-connect** se connecte au serveur Web sur le port HTTP (n° 80). Pour dialoguer avec le serveur à travers ce port, il faudra s'en tenir au protocole HTTP [*HyperText Transfer Protocol*]. Cette fonction retourne deux valeurs (!), des ports d'E/S :

```
> p-in
#<input-port>
```

```
> p-out
#<output-port>
```

• Dans le "Racket Reference", la documentation de la fonction `tcp-connect` dit que les ports `p-in` et `p-out` obtenus sont ***block-buffered*** !

> *Initially, the returned input port is block-buffered, and the returned output port is block-buffered. Change the buffer mode using* ***file-stream-buffer-mode****.*

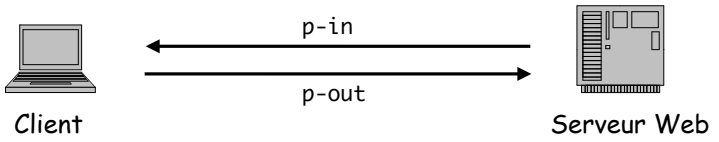
• Cela signifie que les données sont accumulées dans un **tampon** [***buffer***] et envoyées sur le réseau à partir d'un certain volume...

• En pratique, il faut s'occuper de `p-out` :

```
(file-stream-buffer-mode p-out 'none)
```

• Dont acte !

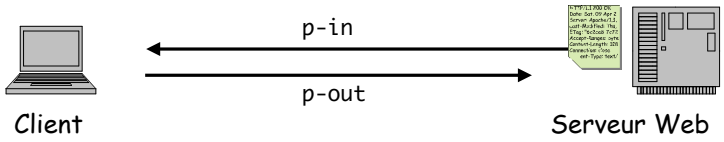
http://docs.racket-lang.org/reference/



• Demandons l'accès à la page `xxx.html`. Nous postons une **requête GET** dans la syntaxe du **protocole HTTP 1.0**. Une requête peut prendre plusieurs lignes mais doit se terminer par une ligne vide :

```
(fprintf p-out
  "GET http://www.boncinema.fr/...206892.html HTTP/1.0\n\n")
```

• Le serveur Web prépare la page pour `p-in` mais ne l'envoie pas :



• Il suffit de faire des `(read-line p-in)` jusqu'à ***eof*** pour lire le fichier ligne à ligne !

• Attention, la page html commence par un **header** [en-tête] de plusieurs lignes renseignant le contenu actuel du fichier. Le *header* se termine par une ligne vide, puis le contenu réel du fichier commence...

EXEMPLE

• Supposons que l'on cherche la *ligne magique* du header (invisible à l'utilisateur !) contenant le mot Content-Type :

```
(define MAGIC-LINE
  (read-line-until p-in
    (λ (str) (regexp-match "Content-Type" str))))
```

read-line-until *en TP !*

⇒ `"Content-Type: text/html; charset=utf-8\r"`

• De manière générale, on pourra trouver une information textuelle dans une page html **si on sait la localiser par une regexp** !

MAJ. 2020 : le code des pages suivantes est meilleur !

```
(define URL
   "http://www.boncinema.fr/film/206892.html")

(printf "***** Connexion au Web sur le serveur www.boncinema.fr\n")

; tcp-connect renvoie deux valeurs (des multiple values) :
(define-values (p-in p-out)
  (tcp-connect "www.allocine.fr" 80))

(file-stream-buffer-mode p-out 'none)    ; pas de buffer en sortie

(fprintf p-out (format "GET ~a HTTP/1.0\n\n" URL))    ; ma requête

(define MAGIC-LINE
  (read-line-until p-in
    (lambda (str) (regexp-match "Content-Type" str))))

(close-input-port p-in)
(close-output-port p-out)

(printf "MAGIC-LINE (dans le header) : ~s\n" MAGIC-LINE)
```

read-line-until *en TP !*

Programmer un client Web HTTP(s) ! new style

• Les développeurs Racket suivent de près l'évolution des techniques et sont prompts à racketter tout ce qui passe. On trouve un module **net/url** permettant de manipuler les URL (protocole RFC 2396) et utiliser le protocole HTTP (**http://**, **https://** et parfois **file://**).

```
(require net/url)
```

• Je veux savoir qui est numéro 4 mondial actuellement au tennis. Je sais que l'information est située sur le serveur `www.tennisclass.net` et plus précisément sur la page au protocole HTTPS :

```
(define URL
    "https://www.tennisclass.net/classement.html")
```

• *Je pourrais utiliser des URL plus compliquées (ici pas besoin) :*

```
http://sky@www:801/cgi-bin/finger;xyz?name=shriram;host=nw#top
{-1}   {2} {3} {4}{---5---------} {6} {----7------------} {8}
```

• Comme en Python avec sa méthode `f.read()`, je peux récupérer cette page Web dans une très grosse chaîne de caractères :

```
(define bigstring
    (port->string (get-pure-port (string->url URL))))
```

une chaîne ← *un port* ← *une URL* ← *une chaîne*

```
>>> bigstring
"<!DOCTYPE html>\n<html lang=\"fr\"  prefix= ....."
```

55929 caractères !

• Il reste à extraire une expression régulière comme dans la méthode *old style* (p. 23), SAUF qu'un problème intéressant se pose !

• Je cherche le joueur numéro 4 mondial dans le source html de la page et je le trouve sous la forme **4**. Je vois que le nom de ce joueur n°4 sera quelque part entre le mot **4** et le mot ****.

• *Comment extraire la sous-chaîne entre deux mots A et B sachant que B se présente plusieurs fois ? Comment stopper au premier B ?*

• On dit que le méta-caractère de répétition * utilisé pour capturer les caractères entre les deux mots A et B est **glouton** (**greedy**), il va essayer de trouver la plus longue chaîne possible.

```
> (regexp-match "foo.*bar" "for foos and bars the stockbar")
("foos and bars the stockbar")
```

et moi j'attends `"foos and bar"`…

• Il faut utiliser la **négation** dans la doc de Racket qui précise que :

`(?!<regexp>)` *Match if ‹regexp› doesn't match*

• Je vais donc tâcher de ne pas traverser un **** pour ne pas passer au numéro 5 avec son **5**. J'utiliserai donc la regexp :

```
(define r "<b>4</b>((?!<b>).)*>([A-Za-z ]*)</a>")
(define res (caddr (regexp-match r bigstring)))
```

```
>>> res
"MEDVEDEV Daniil"
```