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

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

Cours n°8

char et string

pp. 283-294

Unicode et le type char

- **UNICODE** [*www.unicode.org*] distingue entre un ***caractère*** [entité abstraite] et ses ***glyphes*** [images du caractère].
- Les ***polices de caractères*** contiennent des glyphes, alors que UNICODE contient des caractères...
- Traditionnellement un caractère [char] était codé sur un octet [*byte* = 256 possibilités] : insuffisant pour toutes les langues du monde !
- **UNICODE** étend le codage sur plusieurs octets. Le type char de Racket correspond bien à un caractère [alphabet français, lettre cyrillique, caractère chinois, symbole mathématique].
- Techniquement, l'encodage conseillé est **UTF-8**, pour lesquels les caractères du code ASCII sont codés sur un seul octet.

http://fr.wikipedia.org/wiki/UTF-8

- Les caractères usuels sont notés #\a, #\A, #\é, #\! avec certains caractères donnés par nom : #\space, #\newline, #\tab ...

```
> (char=? #\space #\ )
#t
```

```
> (char? #\a)
#t
```

- Les caractères Unicode sont entrés sous la forme #\uxxxx

```
> #\u56fd
#\国
> (string-utf-8-length "\u56fd")
3
```

国

*Le caractère chinois **guó** 国 [nation, royaume] est donc codé sur 3 octets...*

```
> (printf "\u6211\u662F\u6cd5\u56fd\u4eba\n")
我是法国人
```

*(**Wǒ shì fǎ guó rén** : je suis Français)*

pinyin

http://www.mandarintools.com/chardict_u8.html

Le code d'un caractère

- **Chaque caractère a un code entier dans Unicode** :

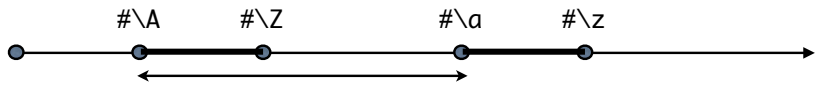
```
> (char->integer #\A)
65
> (char->integer #\a)
97
```

```
> (char->integer #\€)    ; #\u20ac
8364
> (= 8364 #x20ac)
#t
```

- Et ce code est unique :

```
> (integer->char 65)
#\A
> (integer->char 97)
#\a
```

```
> (for ([i (in-range #x0661 (+ 1 #x0669))])
    (write-char (integer->char i)))
١٢٣٤٥٦٧٨٩
```



http://www.unicode.org/charts

Hiragana (Japon)

Les chaînes de caractère

- Une **chaîne de caractères** [*string*] est un tableau de caractères, mais avec une syntaxe spéciale.

```
> (string? "Une chaîne")
#t
```

```
> (string? 'un-symbole)
#f
```

- Une **chaîne constante** [*non mutable*] est définie entre guillemets :

```
> (define str "do re mi fa")
> (immutable? str)
#t
```

```
> (string-length str)
11
```

- Pour la rendre **mutable**, il suffit de la copier :

```
> (immutable? (string-copy "do re mi fa"))
#f
```

- **ATTENTION** : Inclusion d'un " dans une chaîne : `"abcd\"efg"`
Inclusion d'un \ dans une chaîne : `"abcd\\efg"`

Construction d'une chaîne

- Construction *en extension* : `(string char1 char2 ...)`. Le résultat est une chaîne non mutable !

```
> (string #\Y #\o #\u #\space #\w #\i #\n #\!)
"You win !"
```

- Construction *en compréhension* : `(build-string n f)`. Le résultat est une chaîne mutable.

```
> (build-string 10 (lambda (i) (integer->char (+ i 48))))
"0123456789"
```

- *Concaténation* de chaînes : `(string-append str1 str2 ...)`

```
> (string-append "En voici une " "et une autre")
"En voici une et une autre"
```

- Extraction d'une *sous-chaîne* : `(substring str i j)`

```
> (substring "ABCDEFGHIJK" 2 5)
"CDE"
```

[i,j[

Accès aux caractères d'une chaîne

- **Accès** au caractère numéro k [0 ≤ k < n] d'une chaîne de longueur n :

O(1)

```
> (string-ref "abcdefghijk" 5)
#\f
```

`(string-ref str k)`

- **Mutation** du caractère numéro k [0 ≤ k < n] d'une chaîne *mutable* :

O(1)

```
> (define str (string-copy "Le langage C"))
> (string-set! str (- (string-length str) 1) #\B)
> str
"Le langage B"
```

`(string-set! str k c)`

- ***Exemple*** : recherche de la position d'un caractère dans une chaîne :

```
(define (index-of str c)     ; indice de la premiere occurrence de c
  (define (iter i)
    (cond ((>= i (string-length str)) -1)      ; -1 si absent
          ((char=? (string-ref str i) c) i)
          (else (iter (+ i 1)))))
  (iter 0))
```

Et un return comme en Java/C ?

• Il n'y a pas de mot **return** en Scheme, pas plus que de mot **while**. On peut les définir soi-même ! Mais return est plus difficile (L3) !...

• Voici une primitive d'**échappement** un peu délicate, mais bien pratique :

`(let/ec return e1 e2 ...)`

```
(define (index-of str c)
  (let/ec return        ; le mot return est arbitraire !
     (for ([i (in-range (string-length str))])
        (when (char=? (string-ref str i) c)
           (return i)))
     (return -1)))
```

```
> (index-of "abcdef" #\e)
4
```

```
> (index-of "abcdef" #\E)
-1
```

N.B. ***let/ec*** *fait partie des primitives sur les* ***continuations*** *de Scheme, dont la principale est* ***call/cc****. Secteur très avancé et difficile !*

```
def index_of(str,c) :
    for i in range(len(str)) :
        if str[i] == c :
            return i
    return -1
```

Python

```
(define (index-of str c)
  (let/ec return        ; le mot return est arbitraire !
     (for ([i (in-range (string-length str))])
        (when (char=? (string-ref str i) c)
           (return i)))
     (return -1)))
```

Exemple 1 : le codage de César

• Uniquement des lettres et des chiffres ! On produit une copie d'une chaîne en décalant les caractères *alphanumérique* majuscules de k positions vers la droite :

```
(define (cesar str k)    ; décalage de k positions vers la droite
  (define code_A (char->integer #\A))
  (define (code-char c)
    (if (not (char-upper-case? c))
        c
        (let ((dist (- (char->integer c) code_A)))
          (integer->char (+ code_A (modulo (+ dist k) 26))))))
  (build-string (string-length str)
                (λ (i) (code-char (string-ref str i)))))
```

(char)('A' + (c - 'A' + k) % 26)
en Java...

```
> (cesar "message : ATTAQUEZ CE SOIR !" 4)
"message : EXXEUYID GI WSMV !"
```

```
def build_string(n,f) :
    res = ''
    for i in range(n) :
        res = res + f(i)
    return res

def cesar(str,k) :
    code_A = ord('A')
    def code_char(c) :
        if not c.isupper() :
            return c
        dist = ord(c) - code_A
        return chr(code_A + (dist+k) % 26)
    f = lambda i : code_char(str[i]))
    return build_string(len(str),f)
```

Exemple 2 : le générateur de symboles **gensym**

• La primitive (gensym *symb*) permet de produire des symboles numérotés préfixés par *symb* :

```
> (gensym 'etiq)          > (gensym 'etiq)
etiq101                   etiq102
```

• Programmons-le sous la forme d'une *fermeture* avec compteur privé :

```
(define $gensym
  (let ((n 100))
    (lambda (symb)
      (set! n (+ n 1))
      (string->symbol (string-append (symbol->string symb)
                                     (number->string n))))))
```

• Plutôt que supprimer les caractères de la chaîne, on gère l'indice de la position courante dans une variable globale *i* :

```
(define *i* 0)
```

• La fonction (get-char) retourne le caractère courant pointé par *i* et avance *i* . Elle retourne #f si le pointeur *i* est invalide :

```
(define (get-char)
  (if (>= *i* (string-length STR))       ; invalide ?
      #f
      (let ((c (string-ref STR *i*)))
        (set! *i* (+ *i* 1))
        c)))
```

• La fonction (unget-char) fait reculer d'un cran le pointeur *i* :

```
(define (unget-char)
  (set! *i* (- *i* 1)))
```

Exemple 3 : un analyseur lexical

• But du jeu : on part d'une chaîne contenant une expression arithmétique *infixée parenthésée*. Il s'agit d'en extraire les **tokens** [unités lexicales] un par un.

```
> (define STR "23+7*(4-1234)-5")
> (get-token)
23
> (get-token)
#\+
> (get-token)
7
> (get-token)
#\*
> (get-token)
#\(
> (get-token)
4
> (get-token)
#\-
> (get-token)
1234
> (get-token)
#\)
> (get-token)
#\-
> (get-token)
5
> (get-token)
#f
```

fini !

• La fonction (digit->int c) convertit un chiffre c en entier :

```
(define (digit->int c)        ; c est un chiffre
  (- (char->integer c) (char->integer #\0)))
```

• La fonction (get-token) renvoie le premier *token* disponible et positionne *i* sur le caractère qui suit :

```
(define (get-token)
  (define (get-number n)
    (define d (get-char))
    (cond ((not d) n)
          ((char-numeric? d)
           (get-number (+ (* n 10) (digit->int d))))
          (else (unget-char) n)))
  (define c (get-char))
  (cond ((not c) #f)
        ((member c '(#\+ #\- #\* #\/ #\( #\))) c)
        ((char-numeric? c) (get-number (digit->int c)))
        (else (error "Caractère non reconnu" c))))
```

schéma de Hörner !

- Mise en forme : on privatise !

```
(define (tokenizer STR)
  (define *i* 0)
  (define (get-char)
    ...)
  (define (unget-char)
    ...)
  (define (digit->int c)
   ...)
  (define (get-token)
    ...)
  (define (get-number n)
    ...)
  get-token)
```

- Il resterait maintenant à procéder à une **analyse syntaxique** : travailler sur les tokens pour effectuer une *interprétation* de la chaîne : la calculer, la traduire en un autre langage, etc. Ceci sera traité en 3ème et 4ème année...

Les expressions régulières

- Les expressions régulières [*regexp*] sont analogues en Racket à celle utilisées sous Unix par `egrep`. Une *regexp* est un motif décrivant une chaîne et contenant des opérateurs pour combiner des *regexp* plus petites.

- Plutôt qu'une étude complète [faite ailleurs, par exemple en cours *Système*], nous donnerons une galerie d'exemples :

1. Présence d'un caractère dans une chaîne :

```
> (regexp-match "c" "abcdec")
("c")
> (regexp-match "p" "abcdec")
#f
```

- Attention, certains caractères sont spéciaux. Par exemple, le point dans une *regexp* dénote n'importe quel caractère. Pour le "protéger", on le met entre crochets :

```
> (regexp-match "." "abcdec")
("a")
> (regexp-match "[.]" "abcdec")
#f
```

- La primitive `regexp-match*` donne la liste des solutions :

```
> (regexp-match* "." "abcdec")
("a" "b" "c" "d" "e" "c")
```

2. Présence d'une sous-chaîne dans une chaîne :

```
> (regexp-match "[.]jpg" "Les images foo.jpg et bar.jpg")
(".jpg")
> (regexp-match* "...[.]..." "Les images catfoo.jpg et bar.gif")
("foo.jpg" "bar.gif")
```

3. La notation des intervalles [range] :

```
> (regexp-match "[a-z]" "THX-sound 2405")
("s")
```

Si un intervalle débute par un ^ c'est une négation :

```
> (regexp-match "[^A-Z]" "THX-sound 2005"))
("-")
> (regexp-match "[^a-zA-Z-]" "THX-sound 2005")
(" ")
```

4. La répétition par * [0 ou plus] et par + [1 ou plus] :

```
> (regexp-match "[a-z]+" "THX-sound 2005")
("sound")
> (regexp-match "[a-zA-Z]*[.]jpg" "Mais foo.jpg est rouge")
("foo.jpg")
> (regexp-match* "[a-zA-Z]*[.]jpg" "foo.jpg et bar.jpg !!!")
("foo.jpg" "bar.jpg")
```

5. La disjonction par la barre verticale [le ou] :

```
> (regexp-match "0|1" "45b1d08")
("1")                      ↑
```

6. Le début d'une chaîne s'indique par un ^ en tête, la fin par un $ en queue :

```
> (regexp-match "^ab" "abcde")
("ab")                   ↑
> (regexp-match "^ab" "cdabe")
#f
> (regexp-match "de$" "abcde")
("de")                      ↑
> (regexp-match "de$" "abdec")
#f
```

7. Le point d'interrogation ? indique la présence ou non du caractère qui le précède :

```
> (regexp-match "colou?r" "color")
("color")
> (regexp-match "colou?r" "colour")
("colour")
```

8. La primitive `(regexp-replace str1 str str2)` ***permet de remplacer une sous-chaîne str1 de str par str2 :***

```
> (regexp-replace "gif|jpg" "lion.gif et jaguar.jpg" "bmp")
"lion.bmp et jaguar.jpg"
> (regexp-replace* "gif|jpg" "lion.gif et jaguar.jpg" "bmp")
"lion.bmp et jaguar.bmp"
```

9. Si dans une regexp, on met des sous-regexp entre parenthèses, cela les numérote et permet de s'en resservir :

```
> (regexp-replace "([a-zA-Z]+) ([0-9]+)"
                  "en janvier 2014 je dois"
                  "\\2 (\\1)")
"en 2014 (janvier) je dois"
```

- Voilà. Le reste, c'est... de la pratique !

- Exemple d'application pratique, l'affichage d'une liste avec un format différent :

```
(require mzlib/string)      ; pour importer expr->string

(define (afficher L)
  (define str1 (expr->string L))
  (define str2 (regexp-replace* "[(]" str1 "["))
  (define str3 (regexp-replace* "[)]" str2 "]"))
  (define str4 (regexp-replace* " " str3 ","))
  (printf "~a\n" str4))

> (afficher '(re (mi fa) sol))
[re,[mi,fa],sol]
```

C'est la notation utilisée par exemple par Prolog et Python...