# Expressions rationnelles *POSIX*
*(regular expressions)*

*Généralités*
*Mise en œuvre dans la* `libc`

Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
`harrouet@enib.fr`
`http://www.enib.fr/~harrouet/`

# But/Principe

- **Comparer une séquence de caractères à un** *motif*
  - Recherche de *mot-clefs*
  - Découpage en *champs*
  - Analyse lexicale, ...
- **Utilisation de caractères spéciaux**
  - Portions de texte explicites
  - Ensembles ou classes de caractères
  - *"Jokers"*
  - Répétitions de sous-motifs, ...
- **Ces** *motifs* **sont des expressions rationnelles (***ER***)**

# Démarche

- **Compilation de l'expression rationnelle**
  - Effectuée une seule fois pour chaque *ER*
  - Analyse de l'*ER* (dans sa forme textuelle)
  - Constitution de structures de données internes ($\simeq$ machine à états)
- **Confrontation à la séquence de caractères**
  - Effectuée pour chaque *ER* et chaque séquence
  - Y-a-t'il correspondance ?
  - Repérage de l'*ER* dans la séquence
  - Repérage de sous-*ER*

# Exemple informel

- ***ER* à reconnaître :** `(abc)(12)(xy)`
  - Constitué de trois sous-*ER*
  - Les parenthèses sont des caractères spéciaux
- **Confrontation à la séquence :** `helloabc12xyworld`
  - Il y a correspondance
  - L'*ER* est repérée en [5;12[ : `abc12xy`
  - Les sous-*ER* sont repérées en
    - [5;8[ : `abc`
    - [8;10[ : `12`
    - [10;12[ : `xy`

# Constitution d'une *ER*

- **Une expression rationnelle :**
  - → une ou plusieurs *branches* séparées par |
    - ◇ Une *branche* :
      - → une ou plusieurs *pièces* concatenées
        - ○ Une *pièce* :
          - → un *atome* éventuellement suivi d'un *indicateur d'occurence*
            - ○ Un *atome* :
              - → ...
            - ○ Un *indicateur d'occurence* :
              - → ...

# Contenu d'un atome

- **Un caractère litéral**
  - Les caractères `^.[$()|*+?{\` sont *spéciaux*
    → on neutralise leur effet avec `\`
  - ex : `a\.b` → `a.b`
- **Les caractères `( )` :** enferment une sous-*ER*
- **Le caractère `.` :** représente un caractère différent de `\n`
- **Le caractère `^` :** représente le début d'une ligne
  - ex : `^abc` → `abc` à condition que ce soit en début de ligne
- **Le caractère `$` :** représente la fin d'une ligne
  - ex : `abc$` → `abc` à condition que ce soit en fin de ligne
- **Les caractères `[ ]` :** enferment une classe de caractères ...

# Définition d'une classe de caractères

- ▷ `[ ... ]` : représente **UN** caractère parmi tous ceux énumérés
  - ◇ ex : `[ab][12]` → `a1` , `a2` , `b1` ou `b2`
  - ◇ Plages de caractères indiquées par `-`
    ex : `[0-9a-fA-F]` → **UN** chiffre hexadécimal
  - ◇ Les caractères spéciaux perdent leur rôle sauf `-` et `]`
    → placer `]` au début et `-` au début ou à la fin des `[ ... ]`
- ▷ `[^ ... ]` : représente **UN** caractère différent de ceux énumérés
  - ◇ ex : `[^_a-zA-Z]` → un caractère qui ne soit ni une lettre, ni un `_`
- ▷ `[: ... :]` : fait référence à une classe prédéfinie de caractères
  - ◇ `alnum alpha blank cntrl digit graph lower print punct space upper xdigit` (voir `<ctype.h>` : `isalpha() ...`)
  - ◇ ex : `[^_[:alnum:]]` = `[^_[:alpha:][:digit:]]` ≃ `[^_a-zA-Z0-9]`

# Les indicateurs d'occurence

- ▷ **Un atome seul :** Une fois et une seule cet atome
- ▷ `A?` : Zéro ou une fois l'atome `A`
- ▷ `A+` : Une ou plusieurs fois l'atome `A`
- ▷ `A*` : Zéro, une ou plusieurs fois l'atome `A`
- ▷ `A{n}` : Exactement `n` fois l'atome `A`
- ▷ `A{n,}` : `n` fois ou plus l'atome `A`
- ▷ `A{n,m}` : De `n` à `m` fois l'atome `A` ($n \leq m$)
- ▷ **Ne concernent que l'atome immédiatement précédent !**
  - ◇ ex : `ab{3}` → `abbb` alors que `(ab){3}` → `ababab`

# Quelques exemples d'*ER*

entier décimal simple : `[[:digit:]]+` ou `[0-9]+`

entier octal : `0[01234567]+` ou `0[0-7]+`

entier hexadécimal : `0[xX][[:xdigit:]]+` ou `0[xX][0-9a-fA-F]+`

autre entier décimal : `[123456789][[:digit:]]*|0` ou `[1-9][0-9]*|0`

identificateur : `[_[:alpha:]][_[:alnum:]]*` ou `[_a-zA-Z][_a-zA-Z0-9]*`

commentaire C++ : `//.*`

réel :
```
(([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+))([eE][+-]?[0-9]+)?|
[0-9]+[eE][+-]?[0-9]+
```

chaîne simple : `"([^"]|\\")*"`

# L'*API* POSIX

- **Repose sur quatre fonctions et deux structures**
  - `regcomp()` : compilation d'une *ER*
  - `regerror()` : message d'erreur de compilation
  - `regexec()` : confrontation chaîne/*ER* compilée
  - `regfree()` : destruction de l'*ER* compilée
  - `regex_t` : représente l'*ER* compilée
  - `regmatch_t` : résultat d'une confrontation
  - \+ des constantes pour paramétrer les fonctions
- Déclarations : `<sys/types.h>` et `<regex.h>`
- Implémentation : Dans la `libc` standard
- Documentation : `man 3 regcomp` et `man 7 regex`

# L'*API* POSIX

## ▷ Compilation d'une *ER*

- ◇ `int regcomp(regex_t * preg,const char * regex,int cflags);`
- ◇ Initialise `*preg` avec le résultat de la compilation de `regex`
- ◇ Options `cflags` à combiner par `|`
  - ○ `REG_EXTENDED` : → **toujours présent !**
    Utilisation des *ER modernes*, pas les *basiques* (obsolètes)
  - ○ `REG_NEWLINE` : → **toujours présent !**
    Le `.` n'inclu pas le `\n`, ainsi `^` et `$` fonctionnent
  - ○ `REG_NOSUB` :
    On ne cherche pas à situer l'*ER* dans la séquence analysée, juste un test de correspondance
  - ○ `REG_ICASE` :
    Pas de distinction majuscule/minuscule
- ◇ Résultat : `0` si compilation ok, $\neq$`0` si erreur

# L'*API* POSIX

## ▷ Message d'erreur de compilation

◇ `size_t regerror(int err,const regex_t * preg,`
`char * buf,size_t buf_size);`

◇ La compilation de `*preg` doit avoir retourné `err`

◇ Le message descriptif est placé dans `buf` de taille `buf_size` (tronqué si trop petit)

◇ Résultat : taille du buffer nécessaire pour le message complet

◇ Démarche usuelle :

- invoquer avec `buf` et `buf_size` nuls
- allouer `buf` à la taille indiquée
- invoquer à nouveau avec `buf` et la taille allouée
- utiliser `buf` (affichage ...)
- libérer `buf`

# L'*API* POSIX

- **Libération de l'*ER* compilée**
  - `void regfree(regex_t * preg);`
  - Libere les ressources allouées dans `*preg` lors de l'opération `regcomp()`

# Compilation d'une *ER*

```
#include <sys/types.h>
#include <regex.h>
#include <iostream>
using namespace std;
int main(int argc,char ** argv)
{
if(argc>=2)
  {
  regex_t re;
  int r=regcomp(&re,argv[1],REG_EXTENDED|REG_NEWLINE);
  if(r)
    {
    size_t sz=regerror(r,&re,(char *)0,0);
    char * buffer=new char[sz];
    regerror(r,&re,buffer,sz);
    cerr << argv[1] << " --> " << buffer << endl;
    delete[] buffer;
    }
  else cerr << argv[1] << " --> Compilation OK" << endl;
  regfree(&re);
  }
return(0);
}
```

```
$ ./prog "[0-9]+"
[0-9]+ --> Compilation OK
$
$ ./prog "[0-9+"
[0-9+ --> Invalid regular expression
$
```

# L'*API* POSIX

## ▷ Confrontation chaîne/*ER* compilée

◇ `int regexec(const regex_t * preg,const char * str,`
`size_t nb,regmatch_t * match,int eflags);`

◇ Confronte la chaîne `str` à l'*ER* compilée dans `*preg`

◇ Options `eflags` à combiner par `|`

- `REG_NOTBOL` : `^` ne fonctionne pas
- `REG_NOTEOL` : `$` ne fonctionne pas
- Servent à l'analyse d'une ligne par parties successives

◇ Les informations de repérage des *ER* et des sous-*ER* sont placées dans `*match` de taille `nb`
(`nb` vaut généralement `preg->re_nsub+1` ou `1`)

◇ Résultat : `0` si correspondance, $\neq$`0` sinon

# L'*API* POSIX

▷ **Repérage des *ER* et sous-*ER***

◇ La structure `regmatch_t` contient deux indices relatifs à `str` :

○ `rm_so` et `rm_eo` : début et fin de correspondance

○ `-1` si pas de correspondance

◇ Une *ER* compilée `re` contient `re.re_nsub` sous-*ER*

◇ `regexec()` initialise `match` si `REG_NOSUB` est absent de `regcomp()`

○ `match[0]` : *ER* globale

○ `match[1]` : première sous-*ER* (première `(` rencontrée)

○ `match[re.re_nsub]` : dernière sous-*ER* (dernière `(` rencontrée)

◇ Si `match` de taille `1` → repérage de l'*ER* uniquement

◇ Si `match` de taille `re.re_nsub+1` → repérage de toutes les sous-*ER*

◇ Voir l'exemple introductif en page 4

# Test de correspondance — 1/2

```
#include <sys/types.h>
#include <regex.h>
#include <iostream>
using namespace std;
int main(int argc,char ** argv)
{
if(argc>=3)
  {
  regex_t re;
  int r=regcomp(&re,argv[1],REG_EXTENDED|REG_NEWLINE|
                            REG_NOSUB); // no location info
  if(r)
    {
    size_t sz=regerror(r,&re,(char *)0,0);
    char * buffer=new char[sz];
    regerror(r,&re,buffer,sz);
    cerr << argv[1] << " --> " << buffer << endl;
    delete[] buffer;
    }
```

```
$ ./prog "[0-9]+" "abc123xyz"
[0-9]+ matches abc123xyz
$
$ ./prog "[0-9]+" "abcxyz"
[0-9]+ does not match abcxyz
$
```

# Test de correspondance — 2/2

```
  else // regcomp() success
    {
    if(regexec(&re,argv[2],0,(regmatch_t *)0,0)) // 0 regmatch_t required
      {
      cerr << argv[1] << " does not match " << argv[2] << endl;
      }
    else
      {
      cerr << argv[1] << " matches " << argv[2] << endl;
      }
    }
  regfree(&re);
  }
return(0);
}
```

# Repérage de la correspondance — 1/2

```
#include <sys/types.h>
#include <regex.h>
#include <string.h>
#include <iostream>
using namespace std;
int main(int argc,char ** argv)
{
if(argc>=3)
  {
  regex_t re;
  int r=regcomp(&re,argv[1],REG_EXTENDED|REG_NEWLINE);
  if(r)
    {
    size_t sz=regerror(r,&re,(char *)0,0);
    char * buffer=new char[sz];
    regerror(r,&re,buffer,sz);
    cerr << argv[1] << " --> " << buffer << endl;
    delete[] buffer;
    }
```

```
$ ./prog "[0-9]+" "abc123xyz"
[3;6[ --> 123
$
$ ./prog "[0-9]+" "abcxyz"
[0-9]+ does not match abcxyz
$
```

# Repérage de la correspondance — 2/2

```
  else // regexec() success
    {
    regmatch_t m;
    if(regexec(&re,argv[2],1,&m,0)) // only 1 regmatch_t required
      {
      cerr << argv[1] << " does not match " << argv[2] << endl;
      }
    else
      {
      const char * start=argv[2]+m.rm_so;
      unsigned int len=m.rm_eo-m.rm_so;
      char tmp[0x100];
      strncpy(tmp,start,len);
      tmp[len]='\0';
      cerr << "[" << m.rm_so << ";" << m.rm_eo << "[ --> " << tmp << endl;
      }
    }
  regfree(&re);
  }
return(0);
}
```

# Détail de la correspondance — 1/3

```
#include <sys/types.h>
#include <regex.h>
#include <string.h>
#include <iostream>
using namespace std;
int main(int argc,char ** argv)
{
if(argc>=3)
  {
  regex_t re;
  int r=regcomp(&re,argv[1],REG_EXTENDED|REG_NEWLINE);
  if(r)
    {
    size_t sz=regerror(r,&re,(char *)0,0);
    char * buffer=new char[sz];
    regerror(r,&re,buffer,sz);
    cerr << argv[1] << " --> " << buffer << endl;
    delete[] buffer;
    }
```

# Détail de la correspondance — 2/3

```
else // regcomp() success
  {
  unsigned int nb=re.re_nsub+1;
  regmatch_t * m=new regmatch_t[nb];
  if(regexec(&re,argv[2],nb,m,0)) // all sub-expressions required
    {
    cerr << argv[1] << " does not match " << argv[2] << endl;
    }
  else // regexec() success
    {
    for(unsigned int i=0;i<nb;i++)
      {
      if(m[i].rm_so==-1)
        {
        cerr << "<" << i << "> ???" << endl;
        }
      else
        {
```

# Détail de la correspondance — 3/3

```
          const char * start=argv[2]+m[i].rm_so;
          unsigned int len=m[i].rm_eo-m[i].rm_so;
          char tmp[0x100]; strncpy(tmp,start,len); tmp[len]='\0';
          cerr << "<" << i << "> [" << m[i].rm_so << ";" << m[i].rm_eo
               << "[ --> " << tmp << endl;
          }
        }
      }
    }
  regfree(&re);
  }
return(0);
}

$ ./prog "([0-9]+\\.[0-9]*|[0-9]*\\.[0-9]+)([eE][+-]?[0-9+])?" "X12.34e56Y"
<0> [1;8[ --> 12.34e5
<1> [1;6[ --> 12.34
<2> [6;8[ --> e5
$ ./prog "([0-9]+\\.[0-9]*|[0-9]*\\.[0-9]+)([eE][+-]?[0-9+])?" "X12.34Y"
<0> [1;6[ --> 12.34
<1> [1;6[ --> 12.34
<2> ???
```

# Quelques remarques

▷ **Saisie des *ER***

◇ Contenu d'une chaîne ≠ saisie dans le code ou en ligne de commande

○ `char s[]="a\nb\tc";` est équivalent à
`char s[]={'a','\n','b','\t','c','\0'};`

◇ Neutralisation en *C* ou en *Shell* ≠ neutralisation dans l'*ER*

○ Pour `[0-9]+\.[0-9]*` → `char s[]="[0-9]+\\.[0-9]*";`

○ Pour `"([^"]|\\")*"` → `char s[]="\"([^\"]|\\\\\")*\"";`

▷ *ER POSIX* **: minimum disponible partout**

◇ Quelques subtilités supplémentaires ignorées ici

◇ Mises en œuvre dans *egrep*, *awk*, *sed*, *vi* ...

◇ Des variantes plus complètes dans *flex*, *perl*, *java 1.4* ...