Cours 4
Chaînes de caractères

Jean-Jacques.Levy@inria.fr
http://jeanjacqueslevy.net
tel: 01 39 63 56 89

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX
tel: 01 69 33 34 67

http://www.enseignement.polytechnique.fr/informatique/

Plan

1. Caractères – chaînes de caractères
2. Exceptions
3. Entrées-Sortie
4. Filtrage naif
5. Filtrage linéaire
6. Rappel sur les automates finis
7. Expressions régulières
8. Analyse lexicale

Caractères – Chaînes de caractères

Les caractères sont des types primitifs

```
char c = 'a';
```

Caractères spéciaux: `'\n'` (newline), `'\r'` (retour charriot), `'\t'` (tabulation), `'\\` (backslash), `'\''` (apostrophe), `'\"'` (guillemet).

Fonctions: `isLetter(c)`, `isDigit(c)`, `isLetterOrDigit(c)`, etc

Les chaînes de caractères sont des objets

```
String s = "Vive l'informatique!"
```

dont les méthodes

`s.length()` donne la longueur de s
`s.equals(t)` teste l'égalité de s et t
`s.indexOf(c)` donne la première occurence de c dans s
`s.charAt(i)` donne le ième caractère de s
etc.

Les chaînes sont immuables.

Chaînes modifiables

On se sert de la classe `StringBuffer`

```
String x = "a" + 4 + "c"
```

est équivalent à

```
String x = new StringBuffer().append("a").append(4).append("c").toString()
```

La classe `StringBuffer` a deux méthodes

`s.insert(i,t)` insère la chaîne t après le ième caractère de s
`s.append(t)` met t au bout de s

Conversion des chaînes en entier

```
static int parseInt (String s) {
  int r = 0;
  for (int i = 0; i < s.length(); ++i)
    r = 10 * r + s.charAt(i) - '0';
  return r;
}
```

Exercice 1 Le faire en base 16.

Que faire si la chaîne contient des caractères différents des chiffres décimaux?

- Retourner une valeur réservée: -1 par exemple. Laid!
- Lever une exception.

Conversion des chaînes en entier

```
static int parseInt (String s) {
  int r = 0;
  for (int i = 0; i < s.length(); ++i) {
    if (!Character.isDigit (s.charAt(i)))
      throw new Error ("Pas un nombre");
    r = 10 * r + s.charAt(i) - '0';
  }
  return r;
}
```

La classe `Error` est une classe d'erreurs dites “irrattrapables”.
(On n'a pas besoin de le signaler dans la signature de la fonction qui lève cette erreur).

Conversion des chaînes en entier

Avec des erreurs rattrapables:

```
static int parseInt(String s) throws NumberFormatException {
  int r = 0;
  for (int i = 0; i < s.length(); ++i) {
    if (!Character.isDigit (s.charAt(i)))
      throw new NumberFormatException (s);
    r = 10 * r + s.charAt(i) - '0';
  }
  return r;
}
```

`NumberFormatException` **est une sous-classe de la classe générale des** `Exception`**.**

```
java.lang.Object
   +----java.lang.Throwable
           +----java.lang.Error
           +----java.lang.Exception
                   +----java.lang.RuntimeException
                           +----java.lang.IllegalArgumentException
                                   +----java.lang.NumberFormatException
```

Récupération d'exceptions

```
public static void main (String[] args) {
  try {
    int x = parseInt (args[0]);
    System.out.println (x);
  } catch (NumberFormatException e) {
    System.err.println ("Mauvais argument: " + e.getMessage());
  } catch (ArrayIndexOutOfBoundsException e) {
    System.err.println ("Mauvais nombre d'arguments");
  }
}
```

On a séparé le cas des exceptions du traitement normal. A la place de `catch`, on peut écrire `finally` pour effectuer une opération finalle s'il y a exception ou pas (cela évite la duplication de code).

Entrées-Sorties

Impression: `System.out.print`, `System.err.print`
Lecture: 2 constructeurs pour obtenir les entrées tamponnées!

```
import java.io.*;

class IO {
  public static void main (String[] args) {
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    try {
      String s;
      while ((s = in.readLine()) != null) {
        int n = Integer.parseInt(s);
        System.out.println(n);
      }
    } catch (IOException e) {
      System.err.println(e);
    }
  }
```

Exercice 2 Comment éviter d'écrire la clause `catch` ?

Filtrage – *String Matching*

Données: deux chaînes s et t.
But: indice de la première occurence de s dans t.

```
static int match (String s, String t) {
  int m = s.length(), n = t.length();
 loop:
  for (int i = 0; i + m <= n; ++i) {
    for (int j = 0; j < m; ++j)
      if (t.charAt(i+j) != s.charAt(j))
        continue loop;
    return i;
  }
  return -1;
}
```

Complexité?
Cf. l'`animation` de l'université de Kyushu.

Filtrage linéaire – Knuth-Morris-Pratt

Traitement initial sur s pour trouver les plus grands suffixes qui sont aussi des préfixes. On ne recompare pas deux fois un même caractère de t.

```
static int[] preprocess(char[] s, int m) {

  int[] next = new int[m+1];

  int j = 0; int t = -1; next[0] = -1;
  while (j < m) {
    while (t >= 0 && s[j] != s[t])
      t = next[t];
    ++t; ++j;
    next[j] = t;
  }
  return next;
}
```

Complexité?

Filtrage linéaire – Boyer-Moore

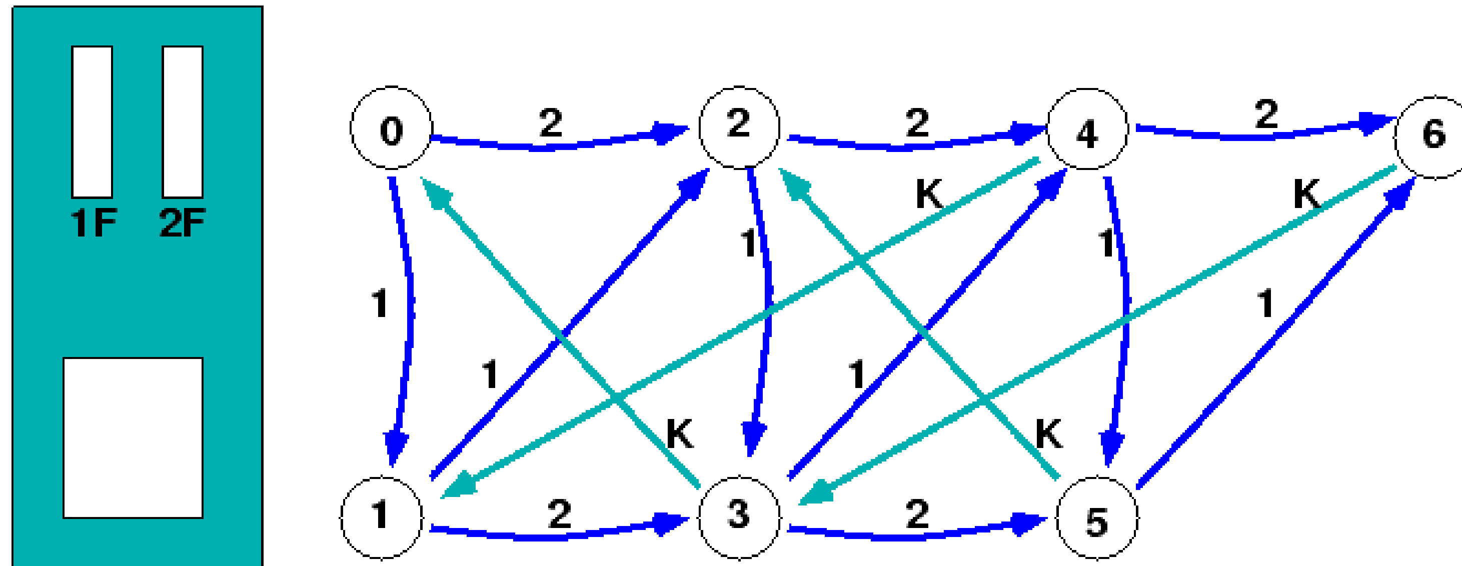
Comme la méthode précédente, mais en procédant de la droite vers la gauche. Sublinéaire.

Quelques notions d'automates finis

- Mécanisme d'un distributeur automatique
- Protocoles réseau (bit alterné, etc)
- Peloton d'exécution (Firing squad problem)
- Systèmes réactifs
- Contrôle dans les circuits
- Analyseurs lexicaux

Distributeur de boissons

Il y a une fente pour les pièces de 1F, 2F. Le café sort si on met 3F. On peut aussi faire une machine avec un tampon de longueur 1 ou 2, qui permet de mettre directement 6F pour avoir 2 cafés.



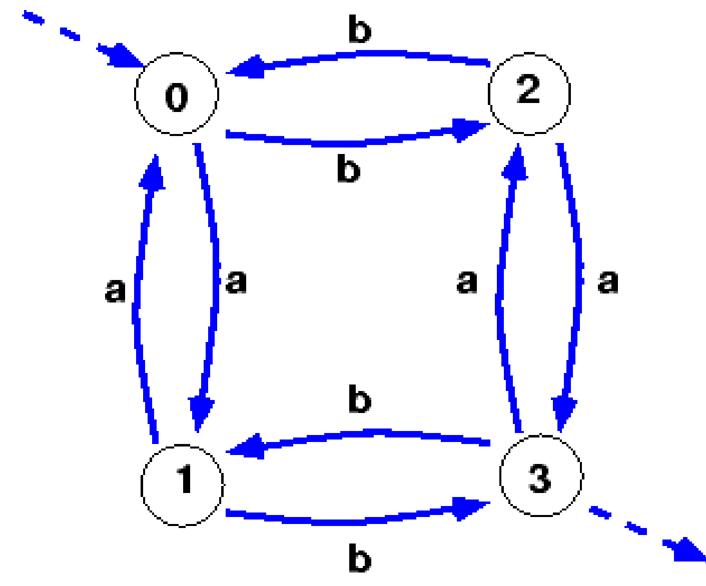
Peloton d’exécution – *Firing Squad Problem*

Il y a une forte brûme sur le plateau. Le général convoque 400 soldats, les aligne, et veut les faire tirer, tous en même temps. A tout moment, le général et les soldats changent d’état de manière synchrone en ne tenant compte que de leurs 2 voisins immédiats. Il y a donc 3 types de machines: le général, les soldats et le soldat du bout de la chaîne.

Montrer qu’on peut trouver une solution en moins de 8 états pour les 3 types de machines quelquesoit le nombre de soldats.

Autre exemple

Automate fini acceptant tous les mots contenant un nombre pair de a et de b.



Automate fini

Un automate fini M est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où:

Σ est un alphabet (de terminaux ou tokens),
Q est un ensemble fini d'états,
$q_0 \in Q$ est l'état initial,
$F \subset Q$ est l'ensemble des états finaux,
$\delta : Q \times \Sigma \rightarrow Q$ est la fonction de transition

On peut étendre δ sur $Q \times \Sigma^* \rightarrow Q$ comme suit:

$\delta(q, \epsilon) = q$
$\delta(q, aw) = \delta(\delta(q, a), w)$

Le langage reconnu par l'automate M est:

$T(M) = \{w \mid \delta(q_0, w) \in F\}$

Graphiquement

Les mots de Σ à analyser sont sur une bande (non modifiable). L'automate est un lecteur de bande. Initialement la tête de lecture est sur la première lettre du mot, l'automate est dans l'état q_0. Puis la tête de bande de déplace d'un cran vers la droite en changeant d'état en fonction du caractère lu, et ainsi de suite jusqu'à la fin du mot. Le mot est accepté si sa fin est obtenue dans un état de F.

Représentation des automates finis

- **En dur dans le contrôle (fini) du programme**

```
int c = in.read(); int q = q0;
switch (q) {
  case 0: if (c == 'a') ... else if (c == 'b)  ...; break;
  case 1: if (c == 'b') ... else if (c == 'c) ... ; break;
  ...
  case 10: if ...  break;
}
```

- **vecteur de $|Q|$ listes d'association (c', q')**

- **matrice $|Q| \times |\Sigma|$ de transition (éventuellement compressée).**

ou

Automates non déterministes

A chaque état, l'automate peut faire un choix non déterministe entre plusieurs transitions. Formellement, $\delta : Q \times \Sigma \rightarrow 2^Q$ et on étend δ comme suit

$$\delta(q, \epsilon) = \{q\} \qquad \delta(q, aw) = \bigcup_{q' \in \delta(q,a)} \delta(q', w)$$

Le langage reconnu par l'automate non déterminite M est:

$T(M) = \{w \mid \delta(q_0, w) \cap F \neq \emptyset\}$

Un mot est accepté si une série de choix peut amener l'automate dans un des états finaux de F. Les mauvais choix ne comptent donc pas.

Exemples non déterministes

- $L = \{xaay \mid x, y \in \Sigma^*\} \cup \{xbby \mid x, y \in \Sigma^*\}$
- $L = L_1 \cup L_2$ **où** L_1 **et** L_2 **sont reconnus par des automates finis**
- $L = \{xwy \mid x, y \in \Sigma^*\}$ **où** w **est un mot donné.**

Déterminisation des automates finis

Théorème 1[Rabin-Scott] Tout langage reconnu par un automate fini non déterministe peut aussi être reconnu par un automate fini déterministe.

Démonstration: on considère l'automate déterministe défini sur 2^Q, en prenant $\{q_0\}$ comme état initial et F comme ensemble de fin.

Remarque: l'automate déterminisé peut avoir 2^n états, si n est le nombre d'états de l'automate non-déterministe.

Théorème 2[Myhill - Nerode] Tout langage reconnu par un automate fini est reconnu par un automate déterministe minimal unique à un isomorphisme près sur le nom des états.

Expressions régulières

Une expression régulière e représente un langage $[\![e]\!]$

$[\![a]\!] = \{a\}$
$[\![e + e']\!] = [\![e]\!] \cup [\![e']\!]$
$[\![ee']\!] = [\![e]\!][\![e']\!]$
$[\![e^*]\!] = [\![e]\!]^*$
$[\![(e)]\!] = [\![e]\!]$

Parfois, on écrit $e \mid e'$ au lieu de $e + e'$.

Exemples

$(a + b)^*abb$ **est l'ensemble des mots sur a et b finissant par abb**
$(a + b)^*w(a + b)^*$ **est l'ensemble des mots contenant w.**

Théorème 3 **Les langages décrits par des expressions régulières sont exactement les langages reconnus par les automates fini.**

Un peu d'algèbre

Ajoutons 0 et 1 en posant $[\![0]\!] = \emptyset$ et $[\![1]\!] = \{\epsilon\}$.

Les lois suivantes sont vérifiées:

$$
\begin{array}{ll}
(1) & e + e' = e' + e \\
(2) & e + (e' + e'') = (e + e') + e'' \\
(3) & e + 0 = e \\
(4) & e + e = e \\
(5) & e(e'e'') = (ee')e'' \\
(6) & 1e = e1 = e \\
(7) & e(e' + e'') = ee' + ee'' \\
(8) & (e + e')e'' = ee'' + e'e'' \\
(9) & 0e = e0 = 0 \\
(10) & 1 + ee^* = e^* \\
(11) & 1 + e^*e = e^*
\end{array}
$$

Expressions régulières et Unix

Les commandes *shell* d'Unix autorisent les expressions régulières.
D'autres commandes comme

`/bin/sh` (Bourne) pour les noms de fichiers,
`sed, ed` (Thomson) Stream editor, Editor
`grep` Get Regular ExPression
Emacs `ESC-x re-search-forward`
`awk, perl` Interpréteurs C (Aho, Weinberger, Kernighan), (Wall)
`lex` (Lesk) Méta-compilateur d'expressions régulières pour C

recherchent des expressions régulières dans les fichiers. Tous ont leur syntaxe propre. En général, ils essaient d'avoir les expressions régulières les plus déterministes possible, sauf `awk` et `perl`.

Regardez le manuel sous Unix

```
% man awk
% man lex
% man sed
```

Analyse lexicale

Comment passer d'une chaîne de caractères à un arbre?

Problème de tout analyseur syntaxique (qu'on verra au cours prochain). Se rappeler de la lecture des arbres dans le cours précédent!

En général, l'analyse syntaxique est précédée d'une phase d'analyse lexicale, qui consiste à retirer des lexèmes dans un flot de caractères.

Lexème: entité importante pour l'analyse syntaxique. Par exemple, un identificateur, une constante numérique, un opérateur.

Les espaces, tabulations, retour à la ligne, commentaires ne sont pas des lexèmes.

Lexèmes et expressions régulières

Ident $=$ Lettre $($Lettre $\mid$ Chiffre$)^*$
Chiffre $= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
Lettre $= a \mid b \ldots \mid z \mid A \mid B \ldots \mid Z$
Entier $=$ Chiffre Chiffre*
Operateur $= + \mid - \mid * \mid /$
Blanc $=$ ' ' $\mid$ \t' $\mid$ '\r' $\mid$ '\n'

Exercice 3 Donner l'expression régulière pour les constantes réelles.

A partir de cette description, on construit l'analyseur lexical en écrivant le programme correspondant à l'automate fini engendré par les expressions régulières. Certains outils (`lex`) peuvent le faire automatiquement. Manuellement, la difficulté réside dans la déterminisation de l'automate.

Exemple d'exécution

Sur la chaîne d'entrée:

```
class PremierProg {
    public static void main (String[ ] args){
        System.out.println ("Bonjour les forts!");
        System.out.println ("fib(20) = " + fib(20));
    }
}
```

le résultat doit être

```
IDENT(class) IDENT(PremierProg) ACCG IDENT(public) IDENT(static)
IDENT(void) IDENT(main) PARG IDENT(String) CROG CROD IDENT(args)
ACCG IDENT(System) POINT IDENT(out) POINT IDENT(println) PARG
CHAINE(Bonjour les forts!) IDENT(System) POINT IDENT(out) POINT
IDENT(println) PARG CHAINE(fib(20) = ) PLUS IDENT(fib) PARG NOMBRE(20)
PARD PARD POINTVIRGULE ACCD ACCD FINFICHIER
```

Représentation des lexèmes

```
class Lexeme {

  final static int NOMBRE=0, IDENT=1, OP=2;
  int nature;
  String as_var; int as_int; char as_op;

  Lexeme (int i) { nature = NOMBRE; as_int = i; }

  Lexeme (String s) { nature = IDENT; as_var = s; }

  Lexeme (char op) { nature = OP; as_op = op; }
```

Et deux fonctions `nextToken()` et une variable globale `curToken`.

En TD la calculette genre HP.