Inf 431 – Cours 5

Analyse lexicale

jeanjacqueslevy.net

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

www.enseignement.polytechnique.fr/informatique/IF

Plan

1. Caractères – chaînes de caractères
2. Exceptions
3. Entrées-Sortie
4. Analyse lexicale
5. Expressions régulières
6. Filtrage

Caractères – Chaînes de caractères (1/2)

- Les caractères ont un type primitif
  `char c = 'a';`
  - Caractères spéciaux :

`'\n'` (*newline*)	`'\r'` (retour charriot)	`'\t'` (tabulation)
`'\\'` (*backslash*)	`'\''` (apostrophe)	`'\"'` (guillemet)

  - Fonctions statiques de la classe `Character` :
    `Character.isLetter(c)`, `Character.isDigit(c)`,
    `Character.isLetterOrDigit(c)`, ...

- Les chaînes de caractères sont des objets de la classe `String`
  `String s = "Vive l'informatique!"`
  - Méthodes de la classe `String` :
    `s.length()` longueur de s,
    `s.equals(t)` égalité de s et t,
    `s.indexOf(c)` première occurence de c dans s,
    `s.charAt(i)` i-ème caractère de s, ...
  - Les chaînes ne sont pas modifiables.

Caractères – Chaînes de caractères (2/2)

- Chaînes de caractères modifiables sont des objets de `StringBuffer`
  `StringBuffer s = new StringBuffer();`
  - Méthodes de la classe `StringBuffer` :

    `s.insert(i,t)` insère la chaîne t après le i-ème caractère de s

    `s.append(t)` met t au bout de s etc.

- Par exemple
  `String x = "a" + 4 + "c";`

  équivaut à

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

Conversion des caractères en entiers (1/2)

- Les caractères ont un code Unicode sur 16 bits (non signés), encore appelé ISO 10646 (سلام, お早う, здравствуй, 你好)
- En Java : char $\subset$ int, mais char $\not\subset$ short.
- Les 256 premières valeurs de l'Unicode sont l'ISO-latin-1 (ISO 8859-1).
- Les 128 premières valeurs sont l'ASCII (*American Standard Codes for Information Interchange*).

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
10	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
20	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	¡	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	ˆ	_
60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	&	}	~	del

Attention : les vieux logiciels ne sont souvent prévus que pour les caractères de code ASCII.

Conversion des caractères en entiers (2/2)

- Compatibilité ascendante du logiciel ⇒ UTF-8
  UTF-8 (*Unicode Transformation Format* - 8 bits)
  = code préfixe en longueur variable.

Unicode	Code UTF-8		
0000 – 007f	`0xxxxxxx`		
0080 – 07ff	`110xxxxx`	`10xxxxxx`	
0800 – ffff	`1110xxxx`	`10xxxxxx`	`10xxxxxx`

Cf. Cours de majeure 2 (info) : Codes et théorie de l'information.

Conversion des chaînes en entiers (1/2)

```
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;
}
```

(En fait, calcul du polynôme $s_0 10^{\ell-1} + s_1 10^{\ell-2} + \cdots + s_{\ell-2} 10 + s_{\ell-1}$ par la méthode de Horner.)

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 entiers (2/2)

```
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;
}
```

`Error` est une classe d'erreurs dites "irrattrapables".

(Pas besoin de les mentionner dans les signatures des fonctions).

Rappel sur les exceptions (1/2)

```
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
```

Rappel sur les exceptions (2/2)

```
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");
  }
}
```

Exceptions $\Rightarrow$ Isolement du cas anormal.

Le cas normal s'écrit indépendamment du cas anormal.

A la place de catch, on peut utiliser finally pour effectuer une opération finale s'il y a exception ou pas (cela évite la duplication de code).

Entrées-Sorties standards

Impression : `System.out.print`, `System.err.print`

```
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);
  }
}
```

l'Inexplicable	`System.in`	flot de bytes
idiome en	`InputStreamReader`	flot de caractères
Java	`BufferedReader`	flot de caractères tamponné

Exercice 2 Comment éviter d'écrire la clause catch dans la fonction *main* ?

Analyse lexicale (1/7)

- But : isoler les lexèmes dans une chaîne de caractères.

- Lexème : entité importante pour une phase ultérieure de calcul.

- Par exemple, un identificateur, une constante numérique, un opérateur, une chaîne de caractères, une contante caractère, etc.

- D'habitude, les espaces, tabulations, retours à la ligne, commentaires ne sont pas des lexèmes.

Analyse lexicale (2/7)

- Entrée : la chaîne de caractères suivante

```
" expression = 3 * x + 2  ; "
```

- Résultat : la suite de lexèmes suivante

```
L_Id(expression) L_Egal L_Nombre(3) L_Mul L_Id(x)
L_Plus L_Nombre(2) L_PointVirgule L_EOF
```

- Même résultat si la chaîne d'entrée est

```
" expression =
        3 * x
      + 2  ; "
```

Analyse lexicale (3/7)

- Entrée : la chaîne de caractères suivante

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

- Résultat : la suite de lexèmes suivante

```
L_Id(class) L_Id(PremierProg) L_AccG L_Id(public) L_Id(static)
L_Id(void) L_Id(main) L_ParG L_Id(String) L_CroG L_CroD L_Id(args)
L_ParD L_AccG L_Id(System) L_Point L_Id(out) L_Point L_Id(println)
L_ParG L_Chaine(Bonjour tout le monde!) L_ParD L_PointVirgule
L_Id(System) L_Point L_Id(out) L_Point L_Id(println) L_ParG
L_Chaine(fib(20) = ) L_Plus L_Id(fib) L_ParG L_Nombre(20) L_ParD
L_ParD L_PointVirgule L_AccD L_AccD L_EOF
```

Analyse lexicale (4/7)

Trois lexèmes (*tokens*) définis par des expressions régulières :

Identificateur = Lettre (Lettre | Chiffre)*
Entier = Chiffre Chiffre*
Operateur $= + \mid - \mid * \mid /$

Lettre $= a \mid b \ldots \mid z \mid A \mid B \ldots \mid Z$
Chiffre $= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
Blancs = (`' '` | `'\t'` | `'\r'` | `'\n'`)*

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

Exercice 4 Donner l'expression régulière pour les constantes chaînes de caractères de Java.

Exercice 5 Donner l'expression régulière pour les constantes caractères de Java.

A partir de cette description, on construit l'analyseur lexical.
Certains outils (`lex`) peuvent le faire automatiquement.

Analyse lexicale (5/7)

```
class Lexeme {

  final static int L_Nombre = 0, L_Id = 1, L_Plus = '+', L_Moins = '-',
        L_Mul = '*', L_Div = '/', L_ParG = '(', L_ParD = ')',
        L_CroG = '[', L_CroD = ']', L_EOF = -1;

  int nature;
  int valeur;
  String nom;

  Lexeme (int t, int i) { nature = t; valeur = i; }
  Lexeme (int t, String s) { nature = t; nom = s; }
  Lexeme (int t) { nature = t; }

  static Lexeme suivant () { ... }
```

Et une fonction `suivant()` qui va chercher le lexème suivant.

Analyse lexicale (6/7)

```
static int c; // caractère courant

static Lexeme suivant() {
  sauterLesBlancs();
  if (Character.isLetter(c)) return new Lexeme (L_Id, identificateur());
  else if (Character.isDigit(c)) return new Lexeme (L_Nombre, nombre());
  else switch (c) {
  case '+': case '-': case '*': case '/':
  case '(': case ')': char c1 = (char)c; avancer(); return new Lexeme (c1);
  default: throw new Error ("Caractère illégal");
} }

static void avancer() {
  try { c = in.read(); } catch (IOException e) { }
}

static void sauterLesBlancs() {
  while ( Character.isWhitespace (c) )
    avancer();
}
```

Attention : revoir la fin de fichier.

Analyse lexicale (7/7)

```
static String identificateur() {
  StringBuffer r;
  while (Character.isLetterOrDigit (c)) {
    r = r.append (c);
    avancer();
  }
  return r;
}

static int nombre() {
  int r = 0;
  while (Character.isDigit (c)) {
    r = r * 10 + c - '0';
    avancer();
  }
  return r;
}
```

Quand il y a de nombreux mots-clé, on donne (comme pour les opérateurs) des numéros de lexème distincts. Chaque lexème identificateur est comparé à une table des mots-clé.

Rappel sur les tables

Table : Clés $\mapsto$ Valeurs
Clés $= \{$`"int"`, `"static"`, `"if"`, `"while"`, $\ldots\}$
Valeurs $= \{$`L_Int`, `L_Static`, `L_If`, `L_While`, $\ldots\}$

Représentations statiques :

- tableau de n Paires (c_i, v_i) $(0 \leq i < n)$ ou
  2 tableaux parallèles Clés et Valeurs de taille n ;
- tableau de n Paires (c_i, v_i) $(0 \leq i < n)$ ordonné sur les clés ;
  recherche dichotomique ou par interpolation ;
- tableau de p listes d'associations avec
  hachage : Clés $\mapsto [0, p-1]$.

Représentations dynamiques :

- liste d'associations $\langle (c_0, v_0), (c_1, v_1), \ldots (c_{n-1}, v_{n-1}) \rangle$;
- arbre de recherche avec nœuds étiquetés par (c_i, v_i) $(0 \leq i < n)$
  ordonné sur les clés.

Expressions régulières (1/5)

Soit un alphabet $\Sigma = \{a, b, \ldots\}$.

Une expression régulière e représente un ensemble $[\![e]\!]$ de chaînes de caractères (de mots), un langage, défini par :

$[\![a]\!] = \{a\}$
$[\![e + e']\!] = [\![e]\!] \cup [\![e']\!]$
$[\![ee']\!] = [\![e]\!][\![e']\!] = \{xy \mid x \in e, y \in e'\}$
$[\![e^*]\!] = [\![e]\!]^* = \{\epsilon\} \cup e \cup e^2 \cup \cdots \cup e^n \cup \cdots$
$[\![(e)]\!] = [\![e]\!]$

où ϵ est le mot vide (de longueur 0).
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.
Par abus de notation, Σ^* est l'ensemble des mots sur Σ

Expressions régulières (2/5)

Posons $[\![0]\!] = \emptyset$, $[\![1]\!] = \{\epsilon\}$ et $e \leq e'$ pour $e + e' = e'$

Les 13 lois suivantes sont vérifiées et définissent une algèbre de Kleene.

(1) $e + f = f + e$

(2) $e + (f + g) = (e + f) + g$

(3) $e + 0 = e$

(4) $e(fg) = (ef)g$

(5) $1e = e1 = e$

(6) $e(f + g) = ef + eg$

(7) $(e + f)g = eg + fg$

(8) $0e = e0 = 0$ demi-anneau

(9) $e + e = e$

(10) $1 + ee^* = e^*$

(11) $1 + e^*e = e^*$

(12) $f + eg \leq g \Rightarrow e^*f \leq g$ (12') $eg \leq g \Rightarrow e^*g \leq g$

(13) $f + ge \leq g \Rightarrow fe^* \leq g$ (13') $ge \leq g \Rightarrow ge^* \leq g$

12-12' et 13-13' sont équivalents.

Pas de théorie équationnelle finie. Pas de formes normales.

Expressions régulières (3/5)

Exercice 6 Montrer $e = e'$ ssi $e \leq e' \leq e$.

Exercice 7 Montrer $e \leq e'$ ssi $e + f = e'$.

Exercice 8 Montrer les équations suivantes :

$$
\begin{array}{rcl}
a^*a^* & = & a^* \\
a^{**} & = & a^* \\
(a^*b)^*a^* & = & (a+b)^* \\
a(ba)^* & = & (ab)^*a \\
a^* & = & (aa)^* + a(aa)^*
\end{array}
$$

Exercice 9 Montrer ces équations en n'utilisant que les règles (1-13).

Pour la première : $1 + aa^* = a^*$ par A10. D'où $aa^* \leq a^*$. D'où $a^*a^* \leq a^*$ par A12'.

Réciproquement, $a^*a^* = (1 + aa^*)a^*$ par A10. Donc $a^*a^* = a^* + aa^*a^*$ par A7 et A5. D'où $a^* \leq a^*a^*$. Etc.

Cf. Les 2 cours de majeure 1 (info) ; livre de Kozen (Automata and Computability)

Expressions régulières (4/5)

Les commandes d'Unix contiennent souvent des expressions régulières.

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

Toutes ont leur syntaxe propre. En général, elles essaient d'avoir les expressions régulières les plus déterministes possible, sauf `awk` et `perl`.
Par exemple,

```
% ls *.java .??*
```

liste les fichiers de suffixe `.java` ou de plus de 3 caractères commençant par un point (.).

Exemple

Expressions régulières (5/5)

```
% sed -e 's/^ *$//'
```

remplace les lignes blanches par des lignes vides sur l'entrée standard.

```
% grep 'Contrôle' *
```

imprime toutes les lignes contenant le mot `Contrôle` dans tous les fichiers du répertoire courant.

```
% grep -i 'analyse.*lexicale' *.tex
```

imprime toutes les lignes contenant le mot `analyse` suivi du mot `lexical` dans tous les fichiers de suffixe `.tex`. (L'option `-i` signifie que majuscules et minuscules sont confondues)

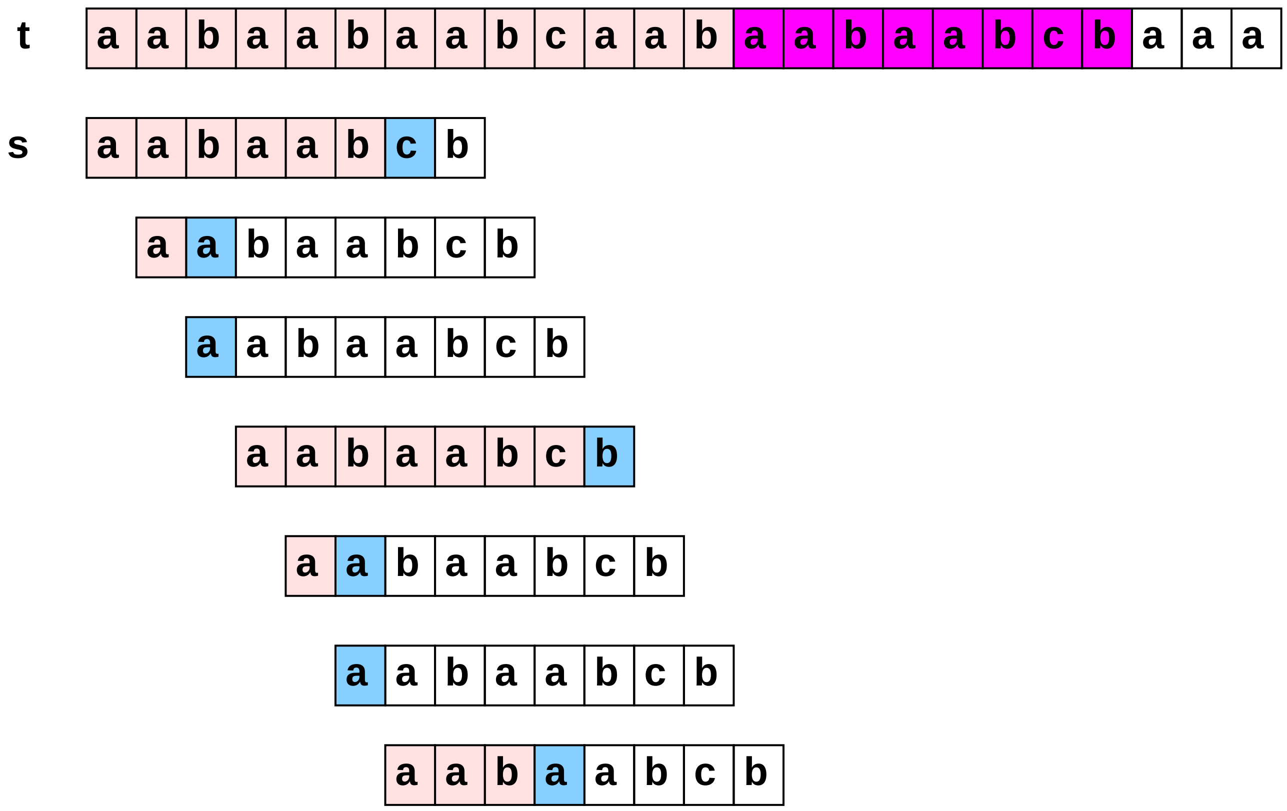
```
% grep '^ *$' a3.tex | wc -l
```

compte le nombre de lignes blanches dans le fichier `a3.tex`

Exemple

Filtrage (1/3)

String Matching : algorithme naif



Complexité en $O(mn)$ où $m = |s|$, $n = |t|$

Animation

Filtrage (2/3)

Filtrage linéaire : [Knuth-Morris-Pratt]

t	a	a	b	a	a	b	a	a	b	c	a	a	b	a	a	b	a	a	b	c	b	a	a	a
s	a	a	b	a	a	b	c	b																
				a	a	b	a	a	b	c	b													
											a	a	b	a	a	b	c	b						
														a	a	b	a	a	b	c	b			

Complexité en $O(m+n)$ où $m = |s|$, $n = |t|$

Animation

Filtrage (3/3)

Filtrage rapide [Boyer-Moore] en procédant de la droite vers la gauche.

t	a	a	b	a	a	b	a	a	b	c	a	a	b	a	a	b	a	a	b	c	b	a	a	a
s	a	a	b	a	a	b	c	b																
				a	a	b	a	a	b	c	b													
							a	a	b	a	a	b	c	b										
										a	a	b	a	a	b	c	b							
													a	a	b	a	a	b	c	b				
														a	a	b	a	a	b	c	b			

Complexité en $O(mn)$ où $m = |s|$, $n = |t|$

souvent sublinéaire.

Animation

Exercices

Exercice 10 En se servant d'une pile, faire une calculette HP (avec la notation polonaise suffixe).

Exercice 11 Modifier l'analyseur lexical pour inclure des commentaires comme en Java, ou en C.

Exercice 12 Idem avec les constantes chaînes de caractères.