INF5171: Laboratoire #5
Parallélisme de flux de données avec filtres et pipelines
OU
Approche «diviser-pour-régner» non récursive

19 octobre 2017
13h30–15h30
PK-S1565

Le but de ce laboratoire est de vous familiariser avec le parallélisme de flux de données et la **décomposition non récursive** d'un problème en sous-problèmes, et ce en utilisant des filtres et pipelines dans le contexte de la bibliothèque `PRuby`.

---

1 Ce que vous devez faire

Obtenez une copie des fichiers pour le labo en exécutant la commande suivante sur `japet` :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/Justification.git
```

Le fichier `enonce.txt` décrit ce que vous devez faire... et sert aussi d'exemple de données pour l'un des cas de test — le résultat attendu est dans le fichier `enonce.resultat`

2 Ce qui vous est fourni

Comme dans les autres laboratoires, divers fichiers vous sont fournis, dont un `makefile`. Notez qu'il n'y a pas de *benchmarks* — l'objectif ici n'est pas d'obtenir du parallélisme mais plutôt d'utiliser la concurrence **pour mieux structurer le programme**, pour bien le décomposer en composants indépendants.

- Fichier `justifier.rb` : Le fichier que vous devez modifier, qui contient des squelettes pour diverses méthodes.
- Fichiers de test `*_spec.rb`.
- Ficher `makefile`, avec des cibles pour chacune des méthodes — voir la constante `WIP` (*Work In Progress*), à modifier au fur et à mesure — en plus d'une cible (`tests`) pour l'ensemble des tests.

3 Quelques informations utiles (méthodes `Ruby` ou `PRuby`)

- La méthode `PRuby::Pipeline.source` crée un noeud source, donc sans canal d'entrée mais avec un canal de sortie. Ce noeud va produire, sur son canal de sortie, une série d'éléments provenant de l'objet fourni en argument, qui peut être n'importe quel objet pouvant répondre au message `#each`, sinon au message `#each_char`, sinon être un nom de fichier. Dans ce dernier cas, ce sont les lignes du fichier qui seront émises, l'une après l'autre, sur le canal de sortie. Toutefois, pour qu'une `String` soit considéré comme un nom de fichier, il faut que le deuxième argument fourni à `Pipeline.source` soit le symbole `:filenmame`.

  Donc :

  ```
  # Emet sur le canal de sortie les caracteres 'f', 'o', ..., 'x', 't'.
  PRuby::Pipeline.source( "foo.txt" )

  # Emet sur le canal de sortie les lignes du fichier 'foo.txt'  (s'il existe)
  PRuby::Pipeline.source( "foo.txt", :filename  )
  ```

- La méthode `get` appelée sur un canal (`Channel`) obtient l'élément en tête du canal **et le retire du canal**. Par contre, la méthode `peek` obtient l'élément en tête du canal **mais sans le retirer**. Les deux méthodes sont bloquantes — on bloque jusqu'à ce qu'une valeur soit émise sur le canal avec `put` (synonyme de `put` = «<<»).

- Les lignes provenant d'un fichier contiennent, en fin de ligne, le caractère «\n». Pour supprimer ce caractère, on utilise la méthode `chomp` (ou `chomp!`), qui n'a aucun effet si ce caractère n'est pas présent :

  ```
  >> a = "abc\n"
  => "abc\n"
  >> a.chomp
  => "abc"
  >> a
  => "abc\n"
  >> a.chomp!
  => "abc"
  >> a
  => "abc"
  ```

- Pour supprimer les blancs en début/fin de chaine, on utilise `strip` (ou `strip!`) :

  ```
  >> "abc ".strip
  => "abc"
  >> "  def abc   ".strip
  => "def abc"
  ```

- Pour déterminer si `ligne` ne contient rien ou ne contient que des blancs, on peut utiliser l'expression de *pattern-matching* suivante :

  ```
  ligne =~ /^\s*$/
  ```

  Pour déterminer si `ligne` débute par le caractère `c`, on utilise l'expression de *pattern-matching* suivante :

  ```
  ligne =~ /^c/
  ```

- Pour déterminer si une chaine `ch` est complètement vide, on utilise `c.empty?` (équivalent à `c == ""`).

- La méthode `join` sur un `Array` de `String` concatène les chaines les unes à la suite des autres, en utilisant le séparateur indiqué entre les items :

  ```
  >> ["abc", "def", "ghi" ].join( ":!:" )
  => "abc:!:def:!:ghi"
  >> ["abc", "def", "ghi" ].join( " " )
  => "abc def ghi"
  >> [].join( "XX" )
  => ""
  ```

- La méthode `split` décompose une chaine en sous-chaines en fonction du séparateur indiqué en argument, retournant un `Array` des sous-chaines obtenues :

  ```
  >> "abc def    gh    ".split(" ")
  => ["abc", "def", "gh"]
  >> "abc".split("")
  => ["a", "b", "c"]
  >> "abc def !!..gh..".split("!")
  => ["abc def ", "", "..gh.."]
  ```

- L'opérateur «`*`» utilisé avec un `String` et un `Fixnum` répète la chaine le nombre de fois indiqué :

  ```
  >> "abc" * 0
  => ""
  >> "abc" * 1
  => "abc"
  >> "abc" * 3
  => "abcabcabc"
  ```

- La méthode «<<» utilisée sur un `String` ajoute le `String` reçu en argument à la fin de la chaine, en la modifiant :

```
>> a = ""
=> ""
>> a << "10"
=> "10"
>> a << " xxx "
=> "10 xxx "
>> a
=> "10 xxx "
```

Ne pas confondre «<<» avec «+», car cette dernière méthod crée une **nouvelle chaine** :

```
>> a = ""
=> ""
>> a + "10"
=> "10"
>> a
=> ""
>>
```