UE 4TIN603U – Compilation – Licence 3 – 2020-2021

TD1 - Écrire un analyseur lexical en JFlex

Les ressources pour débuter cette feuille de td se trouvent sur la page Moodle:

`https://moodle1.u-bordeaux.fr/mod/folder/view.php?id=328477`

1. **Mise en route**

   Vous pouvez choisir de travailler sous Eclipse, ou sous un autre environnement de développement intégré (*IDE*), suivant les recommandations de votre enseignant.
   Si vous choisissez de construire le projet sous Eclipse, lancez *Eclipse* et créez un nouveau projet java `td1` dans votre espace de travail. Sinon, créez un répertoire `td1` dans lequel vous copierez les différents fichiers et répertoires.
   — Créez les répertoires `jflex`, `lib` et `data`.
   — Copiez le fichier `build.xml` à la racine, `pgcd.cpp` dans le répertoire `data`, `lexer.jflex` dans `jflex` et `jflex-full-1.8.2.jar` dans `lib`.
   — Sous `Eclipse`, cliquez avec le bouton droit sur `build.xml` et choisissez `Run as Ant Build`. Sous d'autres environnements, compilez en tapant (dans le répertoire `td1`) la commande `ant` qui va exécuter les directives du fichier `build.xml` (équivalent d'un `Makefile`).
   — Cela va compiler le fichier `lexer.jflex` en un fichier `lexer.java`, puis le compiler en `lexer.class`, puis le lancer sur le fichier `pgcd.cpp` et écrire la sortie sur `output.txt`. `pgcd.cpp` n'est donc pour nous qu'un simple fichier d'entrée pour notre analyseur lexical implémenté dans `lexer.jflex`

2. **Analyseur lexical pour C++ (nous simplifions le langage pour ce TD)** Augmenter l'analyseur lexical de sorte qu'il puisse reconnaître les entités lexicales du langage *C++* qui sont les suivantes :

   (a) Les mots réservés suivants (du moins une partie parmi les plus courants) :

   ```
   bool break case catch
   char class const continue default
   delete do double else enum
   false float for friend goto if inline int long
   namespace new operator private
   protected public register return short
   signed sizeof static struct switch template
   this throw true try typedef typeid typename union unsigned
   ```

```
using virtual void while
```

(b) Les identificateurs, qui commencent par un caractère alphabétique `[a-zA-Z_]` et se poursuivent par d'autres caractères alphabétiques ou des chiffres.

(c) Des constantes entières, qui sont des suites de chiffres.

(d) Des constantes flottantes, qui se composent d'une partie entière, d'un point décimal, d'une partie fractionnaire, d'un e ou d'un E, d'un exposant entier éventuellement signé. Chacune des parties entières et fractionnaires se compose d'une séquence de chiffres. On peut omettre la partie entière ou la partie fractionnaire (mais pas les deux), on peut aussi omettre le point décimal ou le e suivi de l'exposant mais pas les deux.

(e) Les opérateurs, qui sont :

```
++ -- & * + - ~ ! / % << >> < > <= >= == != ^ | && ||
= *= /= %= += -= <<= >>= &= ^= |=
```

(f) Les séparateurs, qui sont :

```
, ; : ( ) [ ] . { }
```

(g) Les constantes de type `string`, qui sont formées d'une suite de caractères entourés de guillemets `"`. Nous simplifions l'encodage en le restreignant au codes ASCII les caractères d'une chaîne.

(h) Les commentaires restreints à une seule ligne, qui débutent par `//`, et les autres commentaires, qui sont précédés de `/*` et suivis de `*/`. Exemple :

```
/****************
* Ceci est un commentaire
*****************/
```

3. **Outil de métrique pour C++**

Afin d'avoir une mesure objective de la quantité de code contenu dans le source (pour servir lors de la facturation par exemple), on modifie encore le projet pour qu'il affiche à la fin quelques données chiffrées :

— Le nombre de lignes
— Le nombre de signes du code, hors commentaires
— Le nombre de mots clefs, d'opérateurs et d'identificateurs