UE 4TIN603U – Compilation – Licence 3 – 2018-2019

TD2 - Mieux comprendre un analyseur lexical écrit en JFlex

Les ressources pour cette feuille de td se trouvent à l'URL suivante:

`https://www.labri.fr/perso/clement/enseignements/compilation/public/td2`

1. **Intégrer l'analyseur lexical**

   L'analyseur lexical (*tokenizer*) que nous avons écrit lors du précédent TD est indépendant de toute application, il permet seulement d'afficher des résultats correspondant aux *tokens* rencontrés dans le fichier source. Nous avions utilisé la commande `JFlex %standalone` pour cela.

   Nous allons maintenant inclure l'analyseur lexical dans une autre application. Le but est de l'intégrer ultimement dans un compilateur complet.

   Pour cela, nous allons utiliser la méthode `yylex()` qui lit le fichier source en parcourant l'automate et renvoie un objet instance de la classe `Token` définie par `%type Token`, ou `null` si la fin de fichier est rencontrée. Pour ne pas perdre du temps avec la programmation Java, nous fournissons la classe `Main` qui effectue la boucle de lecture par appels successifs de `yylex()` :

Listing 1 – Main.java

```
import java.io.FileReader;

public class Main {

    public static void main(String argv[]) {
        if (argv.length == 0) {
            System.out.println("Usage : java Tokenizer <inputfile(s)>");
        }
        else {
            int firstFilePos = 0;
            String encodingName = "UTF-8";
            for (int i = firstFilePos; i < argv.length; i++) {
                Tokenizer tokenizer = null;
                try {
                    java.io.FileInputStream stream =
                        new java.io.FileInputStream(argv[i]);
                    java.io.Reader reader =
                        new java.io.InputStreamReader(stream,
                                                      encodingName);
                    tokenizer = new Tokenizer(reader);
```

```
                    Token token;
                    do {
                        token = tokenizer.yylex();
                        if (token != null)
                            System.out.println (token.toString());
                    }
                    while (token != null);
                }
                catch (java.io.FileNotFoundException e) {
                    System.out.println("File not found : \""
                                       +argv[i]+"\"");
                }
                catch (java.io.IOException e) {
                    System.out.println("IO error scanning file \""
                                       +argv[i]+"\"");
                    System.out.println(e);
                }
                catch (Exception e) {
                    System.out.println("Unexpected exception:");
                    e.printStackTrace();
                }
            }
        }
    }

}
```

Nous vous invitons à suivre pas-à-pas ces procédures :

(a) Créer un énuméré ou une classe `Sym` qui contient la liste des types des *tokens* possibles.

(b) Créer une classe `Token` qui contient les données suivantes :
- `private Sym sym;`, une étiquette permettant de typer le token
- `private Object value;`, la valeur quand le *token* est un litéral
- `private Integer lineno;`, le numéro de la ligne où se trouve le *token*
- `private Integer colno;`, le numéro de colonne où se trouve le *token*

(c) Y implémenter le constructeur `public Token(Sym sym, Object value, int lineno, int colno)`

(d) Y implémenter la méthode `public String toString()` pour afficher le *token*.

(e) Créer le fichier `jflex jflex/Tokenizer.jflex` avec ces commandes :
- `%public`, indique que la classe créée sera publique
- `%class Tokenizer`, indique que la classe crée sera `Tokenizer`
- `%type Token`, indique le type de retour de la méthode `yytext()`
- `%line`, crée l'attribut `yyline` qui contient le numéro de ligne
- `%column`, crée l'attribut `yycolumn` qui contient le numéro de colonne

(f) Poursuivre l'édition de `jflex/Tokenizer.jflex` où chaque *token* reconnu doit correspondre au code `return new Token(...);`

(g) Compiler et tester avec l'exemple `data/sphere.cpp`

2. **Ambiguïtés**

   Augmenter l'analyseur lexical de sorte qu'il puisse reconnaître le mot clef `elseif` en plus des mots clefs du langage `C++`.

   (a) Pourquoi l'analyse de `else` ne masque-t-elle pas celle de `elseif` ?

   (b) L'ordre des deux règles est-elle importante ? Vérifier.

   (c) Vérifier que la règle qui permet de reconnaître les identifiants suit bien celle qui reconnaît les mots clefs. Qu'advient-il si l'on modifie cet ordre ? Pourquoi ?

3. **États**

   Il s'agit maintenant d'analyser les commentaires de notre document `C++` et non plus de les ignorer. En effet, à l'intérieur de commentaires introduits par `/**` nous avons des mots clefs qu'il s'agit d'analyser pour la production de la documentation (dont on fait l'hypothèse qu'elle est réalisée par le compilateur, ce qui est souvent faux).

   Pour cela, nous distinguerons trois états :

   — `<YYINITIAL>` l'état initial et par défaut du compilateur
   — `<COMMENT>` l'état qui passe les commentaires classiques
   — `<COMMENT_DOC>` l'état qui analyse les commentaires lors de la production de la documentation

   (a) Réécrire `jflex/Tokenizer.jflex` en utilisant la commande `%state`

   (b) Modifer pour que l'analyseur prenne en compte les mots clefs `@author`, `@version`, `@param`, `@return` propres à l'état `<COMMENT_DOC>`

   (c) Modifer encore pour que l'analyseur prenne aussi en compte les lignes entières de ces commentaires et les affiche en fonction des mots clefs qui précèdent.

4. **S'arrêter là ?**

   En faisant l'hypothèse que ces commentaires qui servent à produire une documentation contiennent des structures enchâssées de type `XML` ou `LaTex`, donner un argument pour ne pas les traiter au niveau de l'analyse lexicale, mais syntaxique.