Analyse

2 décembre 2014

L'intégralité des sources de ce projet est disponible dans le dépôt github https://github.com/piotr2b/chinese-huawen. Un guide rapide d'installation y est également disponible.

Ce document de fin de projet en montre différentes facettes. Après une présentation générale du contexte plus détaillée que ce qui avait été rédigée pour le cahier des charges, nous expliquerons certains choix techniques et décrirons l'agencement des classes qui composent ce projet. Nous terminerons par des SCENARII d'utilisation possibles.

Table des matières

1 Présentation générale du contexte

Le site piotr.github.io présente le contexte dans lequel s'inscrit ce projet, la structure des caractère chinois et les différents types d'arbre qu'on peut construire pour un caractère. Cette introduction détaillée fait partie du présent document qu'elle est censée éclairer.

2 Choix techniques

LaTeX L'utilisation de ce logiciel de composition de texte est un choix technique de ce projet qui m'a permis d'en améliorer ma compréhension. Plus développé que son concurrent le plus connu Word, plus puissant qu'un simple format de texte comme Markdown, il est paradoxalement plus simple que ces derniers…une fois correctement configuré. Il est en effet facile de partager la configuration des différents fichiers produits pour ce projet avec un système de préambule modulable grace à `import`. LaTeX offre également un contrôle très fin des caractères Unicode utilisés et permet de varier la police en fonction du bloc auquel appartient un caractère. Les caractères a, 華 et 𐐷 sont dans trois polices différentes pour être sûr que chacun soit affiché dans une fonte qui possède un glyphe lui correspondant.

Gephi toolkit Imaginé et conçu par des étudiants Français de l'université de technologie de Compiègne (UTC), Gephi est un logiciel libre d'analyse et de visualisation de graphes, développé en Java et basé sur la plateforme NetBeans. Très puissant, Gephi propose une bibliothèque pour gérer et afficher des graphes. J'ai préféré construire une structure de graphe qui me soit personnelle mais j'utilise la bibliothèque `gephi-toolkit` pour l'affichage.

Maven Maven est un outil de gestion des dépendances et d'intégration continue. J'ai choisi d'utiliser ce logiciel pour faciliter l'import de bibliothèques externes. Il suffit donc pour partager un projet de transférer les classes et de laisser Maven télécharger toutes les dépendances à partir d'un dépôt central. La seule difficulté peut être le bon paramétrage de Maven ou l'intégration manuelle d'une bibliothèque qui manquerait.

Dans un environnement correctement configuré, il suffit d'un simple `mvn clean package` pour télécharger toutes les bibliothèques et construire un paquetage *.JAR exécutable par une machine virtuelle Java.

Java 8 La dernière version de Java est sortie en septembre, quand ce cours commençait. Elle apporte des nouveautés passionnantes comme quelques monades et surtout les expressions lambda, l'évaluation paresseuse et la programmation fonctionnelle en Java. Ces apports offrent l'opportunité de toucher du doigt et de bien comprendre des concepts tels que les fonctions *stateless*, *stateful*, les fonctions court-circuits ou encore *non-interfering*.

Ainsi, ce code, très standard, est suivi de ses équivalents possibles en Java 8 :

Listing 1 : Somme des éléments d'un tableau en Java 7

```
ArrayList<Integer> numbers = new ArrayList<>();
int sum = 0;

for (int x : numbers) {
⟶sum += x;
}
```

Listing 2 : Une autre manière de faire en java 8

```
int sum = numbers
⟶.stream()
⟶.reduce(0, (x,y) -> x + y);
```

Listing 3 : Somme des éléments supérieurs à trois en java 8

```
int sum = numbers
⟶.stream()
⟶.filter(x -> x > 2)
⟶.reduce(0, Integer::sum);
```

jOOQ Je prévoyais d'enregistrer les données obtenues dans une base. Bien que cela soit utile dans l'absolu, ce projet est conçu pour l'instant pour être afficher un graphe de manière autonome. Une base de données n'est donc pour l'instant pas crucial. J'ai cependant intégré jOOQ (Java Object-Oriented Querying), une bibliothèque qui propose un connecteur léger à une base données et des outils de création de classes.

Cette bibliothèque étend JDBC et utilise ce dernier pour communiquer avec la base. Cependant jOOQ ne propose pas à proprement parler un nouveau langage de manipulation de données mais permet de construire des requêtes SQL en s'appuyant sur des objets construisant à partir d'une base existante. La grande proximité qu'elle entretient avec le SQL permet d'utiliser des fonctionnalités avancées du langage SQL comme les `select` imbriqués, les jointures complexes ou les procédures stockées.

MariaDB MariaDB est un système libre de gestion de base de données, clone de MySQL. La mise en œuvre du langage SQL est très proche de ce que nous avons vu dans un autre cours (INF223) et permet donc d'utiliser ce que j'ai appris. Cependant, ainsi que le mentionne le paragraphe précédent, une base de données n'est pas impliqué dans l'utilisation standard de ce micro-projet.

Evolution du paradigme de programmation D'abord commencé en programmation impérative classique, ce projet a rapidement utilisé des idées de programmation fonctionnelle. Les exemples suivants donnent un aperçu du changement.

On remarque dans le premier listing que l'exécution est purement séquentielle. On pourrait cependant parcourir séquentiellement le tableau et lancer un `Thread` pour analyser chaque ligne mais on se heurterait à des problèmes de cohérence. Ici, `AliasMap` est remplacé par deux dictionnaires : le dictionnaire principal `dictionary` et le dictionnaire des alias `alias`.

Listing 4 : Impératif pur. Premier jet sans `AliasMap`

```
Parser<Node, RowChise> parser;
parser = new Parser<>(files, 25000);
Iterator<RowChise> iterator = parser.iterator();

while (iterator.hasNext()) {
⟶RowChise row = iterator.next();

⟶if (row.getCharacter().contains("灣") || row.getCharacter().
    contains("縲")) {
⟶⟶System.out.print("");
⟶}

⟶Node node = new Node(row.getCharacter(), row.getSequence());

⟶alias.put(node.getCharacter(), node.getId());
⟶dictionary.put(node.getId(), node);

⟶main++;
}
```

On utilise maintenant `AliasMap`. Le fonctionnel est sous-jacent mais pas encore directement visible puisque caché dans `AliasMap`.

Listing 5 : Impératif pur. Avec `AliasMap`

```
Parser<Node, RowChise> parser;
parser = new Parser<>(files, 25000);
Iterator<RowChise> iterator = parser.iterator();

while (iterator.hasNext()) {
⟶RowChise row = iterator.next();

⟶if (row.getCharacter().contains("灣") || row.getCharacter().
    contains("縲")) {
⟶⟶System.out.print("");
⟶}

⟶Node node = new Node(row.getCharacter(), row.getSequence());
```

```
13
14 ⟶try {
15 ⟶⟶aliasMap.put(new Alias<Integer, String>(node.getId(), node
       .getCharacter()), node);
16 ⟶} catch (UndefinedAliasException e) {
17 ⟶⟶e.printStackTrace();
18 ⟶}
19
20 ⟶main++;
21 }
```

L'extrait de code suivant correspond à changer la forme sans toucher au fond. On utilise bien un itérateur mais on applique une action sur chacun de ses éléments en utilisant une syntaxe fonctionnelle. C'est un premier pas qui ne révolutionne cependant pas grand'chose.

Listing 6 : Premier pas de fonctionnel

```
1  Parser<Node, RowChise> parser;
2  parser = new Parser<>(files, 25000);
3  Iterator<RowChise> iterator = parser.iterator();
4
5  iterator.forEachRemaining(x -> {
6
7  ⟶if (x.getCharacter().contains("灣") || x.getCharacter().
       contains("縯")) {
8  ⟶⟶System.out.print("");
9  ⟶}
10
11 ⟶Node node = new Node(x.getCharacter(), x.getSequence());
12
13 ⟶try {
14 ⟶⟶aliasMap.put(new Alias<Integer, String>(node.getId(), node
       .getCharacter()), node);
15 ⟶} catch (UndefinedAliasException e) {
16 ⟶⟶e.printStackTrace();
17 ⟶}
18 ⟶main++;
19 });
```

3 Classes principales

Classes et sous-paquet du paquet `entities` Ces classes sont générées automatiquement par `jooq`. Le schéma de la base de données est contenu dans le fichier

`huawen.sql` disponible dans le dépôt github.

Listing 7 : huawen.sql

```
-- This file might better be executed by user root. It deletes a database then
-- create it again.

drop database if exists huawen;
create database huawen;

use huawen;

create table `sinogram` ( -- caractère
	cp varchar(12) not null, -- insert codepoint as a string or a substitute.
	semantics varchar(256),
	consonants char,
	rhyme char,
	tone int,
	stroke tinyint,
	frequency tinyint, -- we further need to distingish between use in speech or use in other sinograms
	induced boolean not null, -- express whether that sinogram has been added properly or induced
	primary key (cp)
);

create table `allography` ( -- similitude
	cp varchar(12) not null,
	class int not null,
	foreign key (cp) references sinogram(cp),
	primary key (cp, class)
);

create table `structure` ( -- composition
	father_cp varchar(12) not null,
	son_cp varchar(12) not null,
	idc enum('⿰', '⿱', '⿲', '⿳', '⿴', '⿵', '⿶', '⿷', '⿸', '⿹', '⿺', '⿻'), -- null allowed if radical
	ordinal int not null,
	foreign key (father_cp) references sinogram(cp),
	foreign key (son_cp) references sinogram(cp),
	primary key (father_cp, son_cp, idc, ordinal)
);
```

`Main` La classe `Main` est essentielle à un programme Java puisqu'elle distingue les exécutables des bibliothèques. Dans ce projet, elle analyse les arguments et lance les fonctions qui correspondent à ce que demande l'utilisateur.

`Node` Un objet de ce type représente un nœud dans le graphe des sinogrammes. Il a donc une collection de composants (de taille 0 si c'est une clef, 2 ou 3) `ArrayList<Node> leaves`, et deux champs `String character` et `IDC idc` pour indiquer le caractère qu'il représente (s'il est connu) et sa composition (toujours connue).

La méthode de classe `Node parse(Deque<String> sequence, Deque<Node> stack)` analyse une IDS et retourne le nœud correspondant.

`Parser` La raison d'être principale de cet analyseur syntaxique est d'agréger des flux de fichiers et de retourner un flux global contenant des lignes de données de type `RowChise`. Cette dernière classe, qui est secondaire, hérite de la classe `Row` en prévision de plusieurs sources de données qui seraient utilisées. Seuls les fichiers de Chise sont utilisés pour l'instant et les plus de quatre-vingt mille sinogrammes qu'ils contiennent sont amplement suffisants.

`Substrate` Cet objet correspond à un *substrat* sur lequel serait couché des nœuds. Cet objet se démarque d'un simple dictionnaire sur au moins deux plans :

- Cohérence des données : deux nœuds qui représentent par exemple les caractères ⿰AB et ⿱BC partageront des références du même objet A
- Facilité de requête : cet objet permet de rapidement connaître tous les caractères composés d'un caractère donné grace à la méthode `List<Node> getCompounds(Node node)`.

Cet objet est l'héritage conceptuel de l'objet `AliasMap` avec lequel j'ai essayé de faire un dictionnaire générique avec deux clefs. Le mécanisme de `type erasure` utilisé en Java pour mettre en œuvre la généricité m'empêchant de faire ce dont j'avais besoin, j'ai préféré concevoir un objet moins générique et plus pratique.

`AliasMap` La structure de données dans laquelle on stocke les caractères est un dictionnaire qui a pour clef l'IDS d'un caractère (ou son hash). Il faut une structure qui évite complètement les redondances, c'est-à-dire que la forêt des sinogrammes n'ait pas de doublon quel que soit l'ordre dans lequel les caractères sont entrés et quel que soit le détail des IDS.

Trois stratégies d'optimisation des accès mémoires :

- Un tas ;
- Un accès aléatoire ;
- Un arbre couvrant.

Stratégie du tas

- Chaque caractère possède ses propres composants ;
- La taille d'un tas est 3 ;
- Solution tributaire des IDS ;
- Explosion combinatoire pour mettre à jour ;
- Analyse de caractère rapide.

Stratégie de l'accès aléatoire

- Lecture et écriture rapide si vraiment aléatoire ;
- Chaque sinogramme contient un tableau de référence de composants.

Stratégie de l'arbre couvrant

- Hypothèse : faiblement connecté ;
- Euh… mais ce n'est vraiment pratique à parcourir !

Un sinogramme comporte trois références vers ses composants et pour l'immense majeure partie deux références. Si le sinogramme est une entrée de la table on a un caractère, un point de code et une IDS. Si c'est un sinogramme induit on n'a qu'une IDS. Certains sinogrammes d'un pan Unicode (*unicode block*) sont décomposés en sinogrammes qui n'ont pas de décomposition dans ce pan mais se décomposent dans un autre pan. L'ordre de décomposition des sinogrammes n'est pas respecté globalement mais seulement à l'intérieur d'un pan. Bien qu'un point de code et une IDS renvoient à une même réalité, il faut les voir comme deux sous-clefs, facette d'une même clef.

Tous les caractères ont une IDS mais seulement les caractères explicites ont un point de code. Les caractères implicites n'ont pas de point de code, seulement une IDS. L'IDS peut donc être vue comme une clef principale à côté de laquelle on place lorsque c'est possible un point de code. L'organisation interne de `AliasMap` devra donc être un dictionnaire `(K1, V)` avec `K1` une IDS et `V` une référence vers un objet `Node`. Il y a également un dictionnaire `(K2, K1)` avec `K2` un point de code. Le dictionnaire réciproque `(K1, K2)` n'est nécessaire que pour économiser la résolution d'une référence : on peut en effet accéder à `K2` très facilement à partir de `K1` : `K1` → `V.K2`.

L'interface de `AliasMap` est finalement un dictionnaire `(K1, K2, V)`.

D'autres personnes ont développé le symétrique de cet objet : un dictionnaire qui renvoie plusieurs valeurs ; d'autres proposent un dictionnaire qui agrège des clefs mais nulle part je n'ai trouvé plusieurs clefs de types différents pour une valeur.

Pourquoi `AliasMap` n'est pas générique La généricité est indispensable pour le dictionnaire `AliasMap<K1, K2, V>` par exemple dans la fonction `put(K clef, V valeur)` qui intègre un nouvel élément V. Il faut tester si K ≡ K1 ou K2.

Ajouter la généricité en Java a été un *tour de force*[1] Comme le montre l'article gj-oopsla java genericity.pdf. Il y est expliqué que le type des génériques n'est pas présent à l'exécution à cause d'une technique appelé effacement du type (*type erasure*). Au contraire du langage `c#` dont il est pourtant proche, Java utilise les type à la compilation mais les enlève et modifie les objets génériques pour leur ajouter les conversions implicites. Les fichiers `*.class` ne porte donc pas trace d'objet générique. L'exemple suivant, tout simple, montre ce mécanisme implicite :

Listing 8 : Code en Java (après 1.5)

```
List<String> stringList = new ArrayList<String>();
stringList.add("Hello World");
String hw = stringList.get(0);
```

Listing 9 : Code équivalent tel qu'écrit dans les fichiers `*.class` (ou avant 1.5)

```
List stringList = new ArrayList();
stringList.add("Hello World");
String hw = (String)stringList.get(0);
```

C'est pour contourner ce mécanisme que j'ai été obligé d'utiliser dans `AliasMap` des champs de classe instancié par le constructeur.

[1] En français dans un texte anglais.

Listing 10 : Constructeur d'un objet faussement générique

```
public AliasMap(Class<? extends Kmain> kMain, Class<? extends
    Kalias> kAlias, Class<? extends V> value, Node.TreeType type)
     {
⟶if (kMain == null || kAlias == null || value == null) {
⟶⟶throw new NullPointerException();
⟶}

⟶KMAIN = kMain;
⟶KALIAS = kAlias;
⟶VALUE = value;
⟶/* ... */
}
```

`PreviewJFrame` Ce projet respecte la séparation entre code et affichage des données. Cet objet est donc le seul objet graphique ; il affiche, si l'utilisateur l'a demandé, un aperçu du graphe des sinogrammes.

4 Utilisations possibles

Les arguments de la ligne de commande permettent quatre cas d'utilisation pour cet analyseur syntaxique. Il est possible d'entrer des valeurs de deux manières différentes et d'afficher un résultat de deux manières différentes :

Listing 11 : Différentes manières d'utiliser l'exécutable de ce projet

```
java -jar parser-0.0.1-SNAPSHOT.jar -direct ▯?▯▯??▯?? -export
    visual
java -jar parser-0.0.1-SNAPSHOT.jar -direct ▯▯▯▯▯ADEBDE -
    export visual
java -jar parser-0.0.1-SNAPSHOT.jar -files test.txt -output
    visual
java -jar parser-0.0.1-SNAPSHOT.jar -files test.txt -output
    terminal
java -jar parser-0.0.1-SNAPSHOT.jar -files test.txt -output
    files
```