TP JFlex et CUP

1 Introduction

Le but de ce TP est de vous donner les bases nécessaire afin de pouvoir écrire votre parseur en Java. Nous commencerons par rappeler le fonctionnement général d'un parseur, comme le couplage d'un analyseur lexical et d 'un analyseur syntaxique. Ces deux éléments, en général Lex et Yacc, sont déclinés pour chaque langage de programmation. Pour Java, nous utiliserons JFlex et CUP. Les différents documents nécessaires au TP sont disponibles sur la page web suivante : `http://www.lsv.ens-cachan.fr/~sangnier/raytracer.php`

1.1 Analyseur lexical

L'analyseur lexical lit les caractères de l'entrée un à un afin de séparer les *unités lexicales* :

- Séparateurs, commentaires, mots clés.
- Identificateurs, entiers, flottants, symboles ou suites de symboles,

Ces entités correspondent à des expressions régulières. Ainsi les symboles appartiennent à un ensemble fini de caractères, et les identificateurs sont des mots finis de lettres (par exemple). L'analyse lexicale correspond donc à une détection de ces unités lexicales.
Pour cela, l'ensemble (infini) des unités lexicales reconnaissables est donné à l'analyseur lexical par le biais d'une *grammaire* finie. Une grammaire est une description finie des règles de génération des unités lexicales à l'aide d'expressions rationnelles. Cette grammaire distingue donc un nombre fini de classes d'unités lexicales, encore appelées *lexèmes* (token en anglais). On retient ensuite, pour chaque unité lexicale reconnue, sa classe et sa valeur qui seront utilisées dans la construction de l'arbre de syntaxe abstraite.

Exemples :

- L'opérateur d'affectation, de classe `AFF`, correspond à la chaîne :=.
- Le mot-clé "int" (`INT`) correspond à la chaîne *int*.
- Un entier (`DIGIT`) est donné par : $\{0,1,2,3,4,5,6,7,8,9\}^+$.
- Un identificateur (`ID`) est donné par : $\{a,\ldots,z,A,\ldots,Z\}^+$.

Remarque : il y a ici une ambiguïté avec l'unité lexicale *int* qui appartient aux deux classes `INT` et `ID`. C'est l'ordre de définition des classes qui imposera le choix parmi les différentes classes auxquelles appartiennent l'unité lexicale (c'est la classe définie en première qui est retenue). On verra dans la suite des moyens pour lever certaines de ces ambiguïtés.

En pratique, on donnera donc à l'outil d'analyse syntaxique (en général Lex ou ses dérivés : Flex, JFlex, ...) un fichier contenant la grammaire des tokens qu'il doit reconnaître. L'outil construira automatiquement à partir de ce fichier du code dans le langage de programmation qui nous intéresse (Caml, C, C++, Java, ...) qui découpera efficacement l'entrée en une suite d'unités lexicales.

1.2 Analyseur syntaxique

Il s'agit à présent de reconnaître la structure du programme. L'analyseur syntaxique vérifie en particulier que le fichier donné en entrée satisfait les règles grammaticales imposées par la syntaxe définie par l'utilisateur.
Il est important de savoir qu'il existe de nombreuses (au moins 4 !) techniques d'analyse syntaxique. Le choix de la technique dépend de la forme de la syntaxe à reconnaître. Nous n'utiliserons pas ce choix ici, puisque nous imposons l'outil à utiliser. Celui-ci reconnaît un unique type de grammaires pour lequel l'analyse est très efficace en pratique.
Étant donnée une description de la syntaxe du fichier d'entrée donnée à l'aide des lexèmes définis lors de l'analyse syntaxique, l'outil génère automatiquement, et dans le langage de programmation utilisé, du code permettant de vérifier la syntaxe de l'entrée et permettant, pour chaque "phrase" ainsi décodée, d'exécuter l'action correspondante (comme construire le nouvel objet de la classe `objet_geo` défini).

Exemple:
La déclaration d'une variable de type entier est de la forme `INT ID AFF DIGIT`. On peut associer au repérage d'une telle déclaration la création d'une nouvelle variable, de type entier, de valeur la valeur associée au lexème de type `DIGIT`.

2 JFlex

Pour l'analyse lexicale, nous utiliserons l'outil JFlex (`http://jflex.de`).

2.1 Installation de JFlex

Téléchargez l'archive complète de la version 1.4.1 de JFlex en .tar.gz sur la page web du projet. Décompressez ensuite cette archive dans un répertoire permanent (vous trouverez des exemples dans l'archive décompressé). JFlex est déjà installé sur les machines de la salle 411, mais si vous souhaitez l'utiliser sur une autre machine, il est conseillé d'inclure le chemin (`jflex-1.4.1/bin`) dans votre variable de PATH, ce qui permet d'utiliser la commande `jflex`.

2.2 Premier exemple

Un exemple de spécification JFlex qui n'est pas rattachée à une spécification Cup est donné dans l'archive de JFlex que vous avez téléchargée. Il se trouve dans le dossier `jflex-1.4.1/examples/standalone`. Le code du fichier `standalone.flex` est recopié ici:

```
%%

%public
%class Subst
%standalone
%unicode
%{
  String name;
%}

%%
```

```
"name " [a-zA-Z]+  { name = yytext().substring(5); }
[Hh] "ello"        { System.out.print(yytext()+" "+name+"!"); }
```

Lisez la documentation attachée à ce fichier, faites vos premiers tests de compilation avec JFlex, et modifiez le fichier d'entrée afin de tester la programme.

2.3 Format d'une spécification JFlex

Le code d'un fichier JFlex est divisé en trois parties comme indiqué ci-dessous :

```
Code de l'utilisateur
%%
Options et déclarations de macros
%%
Règles lexicales
```

Code de l'utilisateur
Le code de l'utilisateur sera recopié "verbatim" dans l'entête du fichier généré par JFlex. Ce fichier généré sera un fichier java, contenant une unique classe. Essentiellement, le code écrit ici contiendra des chargements de packages et de librairies.

Options et déclarations de macros
Les options permettent de passer des arguments à l'outil JFlex. Quelques-unes sont présentées ci-dessous. Se référer au manuel de JFlex pour plus de détails:

- `%class toto`
  demande à JFlex de nommer le fichier produit "toto.java". La classe contenue dans ce fichier sera elle aussi nommée "toto".
- `%public, %final, %abstract, %implements interface1`
  demande à JFlex de déclarer la classe produite avec l'option correspondante.
- `%cup`
  permet d'utiliser JFlex avec Cup.
- `%line %column`
  permet de compter les lignes et les colonnes dans les variables `yyline` et `yycolumn` respectivement.
- `%standalone`
  permet d'utiliser JFlex seul. Ainsi, la classe générée contiendra une fonction `main`.
- `%state toto`
  déclare un identifiant d'état de l'analyse lexicale. L'utilisation de ces états sera décrite plus bas.

Les déclarations de macros permettent des abréviations dans la définition des règles lexicales. Elles sont de la forme `idmacro1 = ER1`. Ainsi on pourra utiliser par la suite l'expression régulière `ER1` en notant simplement `idmacro1`.

Règles lexicales
C'est ici que l'on définit la grammaire permettant de générer les unités lexicales reconnues dans

notre spécification.
Reprenons l'exemple donné plus haut. Deux règles sont données :

- si le mot clé "name" est détecté, la chaîne de caractères composée de lettres qui le suit est stockée dans la variable `name`. (`substring(5)` permet d'oublier les 5 premiers caractères).
- si un des deux mots clés "hello" ou "Hello" est reconnu, on affiche ce mot-clé suivi du contenu de la variable `name`

Remarque: comme nous sommes en mode "standalone", toute unité lexicale qui ne satisfait aucune des règles est recopiée sur la sortie standard.

Pour une définition exhaustive des expressions régulières dans JFlex, se référer au manuel. Vous en verrez de plus intéressantes (et a priori suffisantes) dans l'exemple suivant.

3 JFlex + Cup

Pour l'analyse syntaxique, nous utiliserons l'outil CUP (`http://www2.cs.tum.edu/projects/cup/`). Vous pouvez trouver une documentation en ligne complète sur Cup en anglais à cette adresse. On donne toutefois les bases minimales dans la suite de ce document.

3.1 Utilisation de Cup

Cup est programmé en Java. Récupérez l'archive jar de CUP sur la page web du projet Java. Cette archive correspond à la version $0.11a$. L'utilisation de cette archive se fait de la façon suivante :
`java -jar java-cup-11a.jar moncup.cup`
(moncup.cup étant le fichier contenant la spécification Cup).
À partir d'un fichier d'entrée respectant le format spécifié ci-dessous, Cup génère deux fichiers : un fichier parser.java contenant la classe permettant de parser par la suite, et un fichier sym.java contenant les types des objets utilisés (symboles).

3.2 Format d'une spécification Cup

Nous présentons le format général d'une spécification CUP sur un exemple :

```
import java_cup.runtime.*;
import java.util.Vector;
import java.io.*;

parser code {:
//public Lexer lex;
:}

terminal LPAR, RPAR, PLUS, MOINS;
terminal String ID, NB, STRING;

non terminal String term_form;
non terminal res;

precedence left PLUS;
```

```
precedence left MULT;

res ::= term_form:t {: System.out.println(t); :}
;

term_form ::=
          term_form:t1 PLUS  term_form:t2 {: RESULT = t1 + "+" + t2; :}
        | term_form:t1 MULT term_form:t2 {: RESULT = t1 + "*" + t2; :}
        | LPAR term_form:t RPAR {: RESULT = "(" + t + ")"; :}
        | ID:id {: RESULT = id; :}
        | NB:nb {: RESULT = nb; :}
;
```

Le code précédent est un extrait du code de l'exemple suivant. On constate que le code est divisé en différentes parties :

- importation de package et de librairies,
- code de l'utilisateur,
- liste des symboles (non) terminaux, éventuellement typés,
- déclaration des précédences,
- les règles de la grammaire.

Le code de l'utilisateur est inclus directement dans le code de la classe parser du fichier parser.java. D'autres classes sont définies dans le fichier parser.java, mais on n'aura en général pas besoin de les modifier.
Les symboles typés sont ensuite utilisés avec le type correspondant, lors des actions associées aux règles de la grammaire.
Les symboles non-typés ne peuvent pas contenir de valeur.
Les règles de précédence sont utilisées pour déclarer des règles de priorité entre opérateurs (exemple : $2 + 3 * 5$). Les opérateurs sont triés par priorité croissante. Ainsi dans notre exemple l'opérateur MULT a une plus grande priorité que l'opérateur PLUS et on obtiendra l'évaluation $(2 + 3) * 5$ Enfin, le mot clé "left" signifie que l'opérateur est associatif à gauche : $1 + 3 + 5$ sera évalué en $(1 + 3) + 5$.
Les règles de la grammaire enfin, sont écrites à l'aide des terminaux et des non terminaux. Par défaut, l'algorithme va chercher à satisfaire la première règle de production. Lorsqu'une règle est satisfaite, l'action associée est réalisée.

3.3 Un exemple plus réel

Récupérez sur la page web du projet Java l'archive contenant un exemple complet de parseur.
Cette archive contient le jar java-cup-11a.jar, et les spécifications JFlex et Cup pour un exemple simple de parseur d'expressions numériques, ainsi qu'un fichier d'entrée d'exemple. Enfin, l'archive contient un fichier Makefile permettant d'automatiser la compilation.
Éditez le fichier Makefile. On remarque de quel façon, on génére la classe `Lexer.java` grâce à JFlex et les classes `sym.java` et `parser.java` en utilisant CUP. Notons de plus, que pour compiler la classe `Lexer.java`, il est nécessaire d'avoir compilé au préalable les classes générées par CUP (en particulier `sym.java` où sont déclarés les différents symboles lexiquaux). De plus pour compiler vos différentes classes .java (avec `javac`) et pour exécuter le programme (avec `java`), vous devrez inclure le chemin vers le jar de CUP en ajoutant l'option `-classpath java-cup-11a.jar:.` comme indiqué dans le Makefile et dans le ficher README.

Dans un premier temps, lisez les deux fichiers de spécification afin de comprendre leur inter fonctionnement et testez le comportement du programme sur quelques exemples.
Nous souhaitons à présent augmenter la capacité de notre programme de sorte à en faire un évaluateur d'expressions comportant des variables et des entiers relatifs.
Modifiez le programme de sorte à calculer la valeur numérique de l'expression donnée en entrée. On supposera dans un premier temps que cette expression ne contient que des entiers (pas de variables).
On souhaite enfin intégrer des définitions de variables. Le fichier d'entrée comportera donc une première partie, dans laquelle nous définirons des variables de type entier ou flottant. Dans une deuxième partie, des définitions d'expressions numériques portant sur ces variables seront données. Le parser devra calculer les valeurs numériques de ces expressions.

Exemple de fichier d'entrée :

```
Variables
int x = -2;
float y = 3.14151;
int z = 0;

Expressions
E1 = (3*x+4*y)*2 -z+1 ;
E2 = (3*z)-2;
```

Commencez par le cas où toutes les variables sont entières puis étendez au cas de variables quelconques.
Pour finir, rajoutez dans votre programme l'opérateur division ainsi que des exceptions levées lors de divisions par 0 ou lors d'utilisation d'une variable non déclarée.