Feuille de travaux pratiques n° 2

Introduction à JFlex

JFlex est un générateur d'analyseurs lexicaux pour Java. Il prend en entrée un fichier décrivant un *système de définitions rationnelles* et une *liste d'expressions rationnelles* associées à des actions Java. En sortie, il produit une classe Java (souvent nommée `Lexer`) contenant une méthode `yylex()` capable de découper un flux de caractères en une suite de *tokens*, c'est-à-dire des sous-chaînes de caractères formant des unités lexicales.

Chaque expression rationnelle est associée à une action Java exécutée à chaque reconnaissance du motif. Cette action peut être une instruction simple, un bloc d'instructions entre accolades, ou bien une instruction de retour (`return`) lorsqu'on veut produire explicitement un objet `Token`. Si aucune action n'est précisée, la chaîne reconnue est simplement ignorée.

JFlex est un outil de génération automatique très utilisé pour la phase d'analyse lexicale des compilateurs, interpréteurs et outils de traitement de texte. Il reprend les principes historiques du programme `lex`, développé dans les années 70 aux laboratoires Bell d'AT&T, mais en génère directement du code Java plutôt que du C.

Dans ce TP, vous allez utiliser JFlex pour construire un analyseur lexical pour le langage algorithmique `Algo`. Les expressions rationnelles utilisées ici reposent sur les notions vues au TP précédent ; nous nous concentrerons sur leur intégration dans un analyseur lexical et sur le comportement du moteur JFlex.

Exercice 1 (Mise en place).

Un exemple de programme utilisant *JFlex* est fourni dans le dépôt. Il servira de base aux exercices suivants. Nous allons commencer par le télécharger, générer l'analyseur et compiler le projet.

1a. Clonez le dépôt :

```
git clone https://github.com/LangagesEtAutomates/lea-tp2-lexer
cd lea-tp2-lexer
```

1b. Générez l'analyseur et compilez le projet :

```
ant compile
```

Remarque : La génération de l'analyseur et la compilation doivent être relancées après chaque modification d'un fichier `.flex`.

1c. Expliquez chaque ligne écrite par la génération de l'analyseur dans le terminal.

1d. Vérifiez qu'un fichier Java a bien été généré dans `gen/lea/Lexer.java`. Vous pouvez ouvrir ce fichier et remarquer la présence d'une méthode `public Token yylex()`, mais n'essayez pas de le comprendre en détails.

1e. Testez le programme. Vérifiez que tous les tests de `Exo1Test` passent, et que les autres tests échouent.

```
ant test
```

1f. Exécutez le programme :

```
ant run
```

Si vous travaillez sous *Eclipse*, vous pouvez lancer les tâches `ant` en utilisant les configurations préconfigurées. Allez dans ; `Run → Run Configurations...`, puis choisissez :

- `Launch Group → Compile-lea-tp2-lexer`
- `Launch Group → Test-lea-tp2-lexer`
- `Java Application → Run-lea-tp2-lexer`

Il est nécessaire de rafraîchir le projet (`F5` sur sa racine) après chaque compilation.

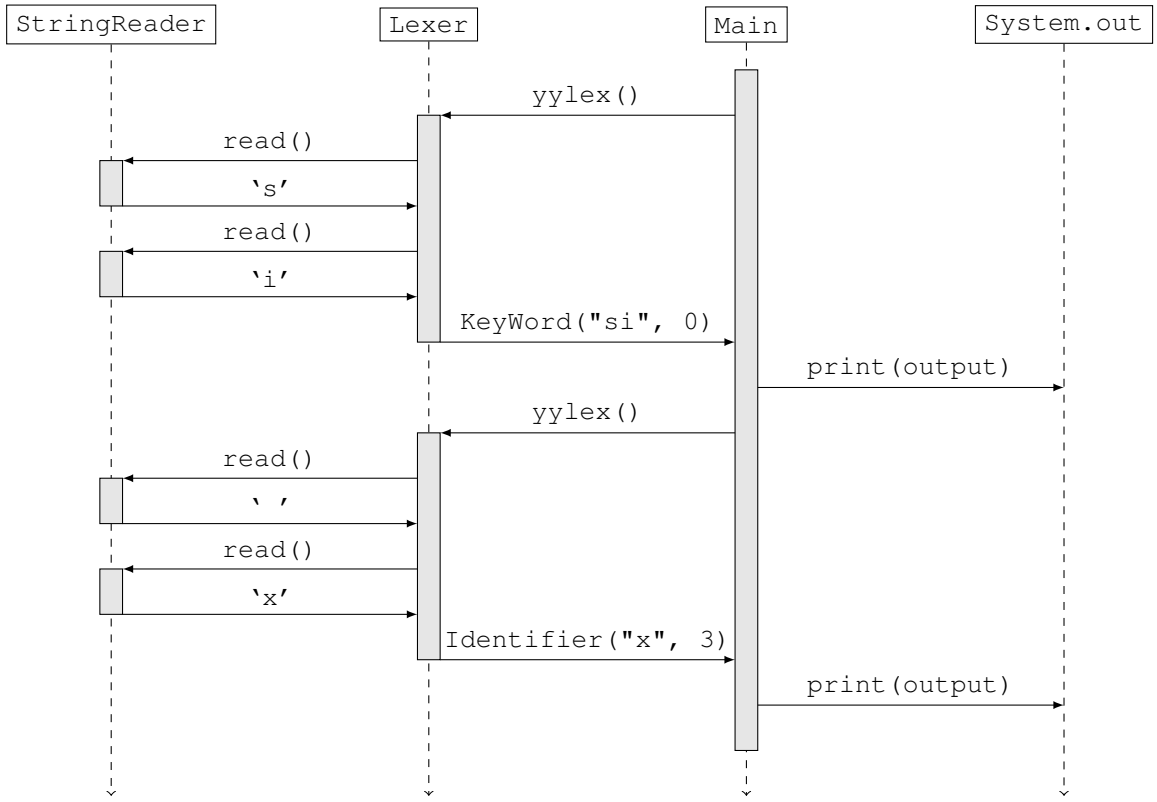
Exercice 2 (Reconnaissance de motifs).

L'objectif de cet exercice est de comprendre comment utiliser le lexer généré par JFlex pour produire et afficher un flux de tokens.

2a. Ouvrez et lisez le fichier `src/lea/Main.java`.

Architecture du programme. L'entrée du programme est la méthode `Main.main`, qui se contente d'appeler la méthode `Main.readString()`. Cette méthode construit un `StringReader` à partir du texte d'entrée, puis crée un objet `lexer` de type `Lexer` (généré automatiquement à partir du fichier `Lexer.flex`). Elle appelle ensuite `lexer.yylex()` en boucle, jusqu'à ce que le lexer signale la fin du texte en retournant **`null`**. La suite des valeurs non nulles retournées par `yylex()` constitue le *flux de tokens* produit par le lexer.

L'architecture du programme est résumée par le diagramme de séquence suivant : `StringReader` lit l'entrée caractère par caractère, `Lexer` reconnaît des motifs et les regroupe en tokens, et `Main` affiche le texte de chaque token coloré sur la sortie standard.



Représentation des tokens. Les valeurs retournées par `Lexer.yylex()` sont des objets de type `Token`. Chaque token est de la forme `token = Type(String text)`, où

— `Type` est l'un des types concrets qui implémentent l'interface `Token`. Plusieurs de ces types sont déjà fournis : ce sont des enregistrements (`record`) déclarés dans le corps de l'interface : `Token.KeyWord` pour les mots-clés, `Token.Number` pour les nombres, etc.

Les `record`s fournissent automatiquement constructeur, accesseurs et égalité structurelle. Le mot-clé `sealed` dans la déclaration de l'interface garantit que seules les variantes explicitement déclarées dans cette interface (comme `KeyWord` et `Number`) peuvent exister.

Cela permet d'utiliser le *pattern matching* dans un bloc **`switch`**, où le compilateur vérifie que chaque type de token est bien traité et transtype automatiquement le token dans une variable locale du bon type.

— `token.text()` contient le *lexème*, c'est-à-dire la sous-chaîne du texte reconnue par un motif.

2b. Ajoutez la ligne suivante dans le corps de l'interface `Token` afin de déclarer un troisième type de tokens, destiné aux identifiants (noms de variables) :

```
public record Identifier(String text) implements Token {}
```

2c. Ouvrez et lisez le fichier `src/lea/Lexer.flex`.

Structure d'un fichier JFlex. La première partie du fichier (avant le premier `%%`) contient du code Java copié tel quel dans le fichier généré. Ici, elle se limite à la déclaration du package.

La seconde partie (entre les deux `%%`) contient les *options de génération* et les *déclarations JFlex*. Dans cet exemple :

— `%public` : la classe générée sera publique ;

— `%class Lexer` : nom de la classe Java générée ;

— `%type Token` : indique que la méthode `yylex()` retourne un objet de type `Token` ;

— `%unicode` : prise en charge correcte des caractères UTF-8.

— `%line` et `%column` : enregistre la ligne et la colonne du début du dernier motif reconnu dans des variables `yyline` et `yycolumn`.

La partie suivante, entre `%{` et `%}`, contient du code Java qui sera ajouté dans le code généré de la classe `Lexer`. Ici, on déclare un champ `reporter` qui sera utilisé pour reporter des erreurs lexicales dans l'analyse des fichiers.

Juste en dessous, on trouve des *définitions rationnelles*, qui donnent un nom à une expression rationnelle. Par exemple, `DIGIT = [0-9]` définit le motif nommé `DIGIT`. Plus bas dans le fichier, le motif qui reconnaît les nombres utilise cette définition en écrivant `{DIGIT}` à la place de `[0-9]`.

Enfin, la troisième partie (après le second `%%`) décrit les *règles lexicales* : chaque ligne associe un motif (*expression rationnelle*) à une action Java entre accolades.

Lorsque plusieurs expressions rationnelles peuvent correspondre à une position donnée du texte, *JFlex choisit le token de plus grande longueur*. S'il existe plusieurs expressions de même longueur, *la première règle déclarée dans le fichier est prioritaire*.

Dans cet exemple :

— les mots `si` et `sinon` sont reconnus comme mots-clés et retournent un token de type `Token.KeyWord` ;

— les suites non vides de chiffres sont reconnues comme nombres et retournent un token `Token.Number` ;

— la règle `[^]` (tout caractère isolé) joue le rôle de « rattrapage » : pour chaque caractère qui ne correspond à aucune autre règle, elle recopie ce caractère sur la sortie d'erreur (`System.err.print(yytext());`). Elle ne retourne pas de `Token` : ces caractères n'apparaissent donc pas dans le flux de tokens.

2d. Ajoutez de nouvelles règles pour reconnaître les opérateurs arithmétiques `+`, `-` et `*`, et retourner des tokens de type `Token.Operator`.

2e. Un identifiant de variable suit la convention classique : une lettre ou _ en premier caractère, puis une suite (éventuellement vide) de lettres, chiffres ou _. Ajoutez la règle suivante *avant* les règles pour les mots-clés, puis déplacez-la *après* celles-ci. Que constatez-vous ?

```
[A-Za-z_][A-Za-z0-9_]* { return new Token.Identifier(yytext()); }
```

2f. Observez la règle qui reconnaît les commentaires multi-lignes : `"/*"([^*] | "*"+ [^/])* "*"+ "/"`
Pourquoi n'a-t-on pas simplement utilisé le motif `"/*". * "*/"`, plus lisible ? Donnez un exemple de texte qui serait reconnu par ce motif alors qu'il ne devrait pas l'être, et un cas où ce motif échouerait alors qu'il devrait réussir.

2g. On souhaite reconnaître des littéraux de caractère entre guillemets simples.

— Déclarez dans l'interface `Token` un nouveau type `CharLiteral`.

— Ajoutez dans `lexer.flex` une règle JFlex qui reconnaît les littéraux de caractère. Un caractère et soit un antislash suivi de n'importe quel caractère (`\\.`), soit un caractère quelconque différent de `'`, `\` et du passage à la ligne (`[^'\\\n]`), le tout entouré par des guillemets simples (`'`).

— Dans `Main` adaptez l'affichage pour colorer les `Token.CharLiteral` en magenta (format ANSI `FG_MAGENTA`). Vérifiez que `'a'`, `'_'` et `'\\n'` apparaissent correctement colorés.

La dernière règle du fichier JFlex est une règle de « rattrapage » : le motif `[^]` correspond à n'importe quel caractère isolé, donc il ne peut être utilisé qu'en dernier, quand aucune règle plus spécifique ne s'applique. Dans le code fourni, cette règle recopie le caractère en gris :

```
[^] { System.out.print(Main.FG_GRAY + yytext() + Main.RESET); }
```

Dans un analyseur lexical, un caractère inconnu ne doit pas être simplement ignoré, mais reporté comme une erreur. On doit donc faire la distinction entre un *caractère correct* mais qui ne joue pas de rôle dans le programme, comme les espaces et les commentaires, et les caractères illégaux. Nous allons modifier cette règle pour qu'elle signale une erreur lexicale via le `Reporter`, mais sans déclencher d'erreurs sur les espaces.

2h. Ajoutez une règle qui reconnaît une séquence non vide d'espaces, tabulations et passages à la ligne (`[ \t\n]+`) et affiche un point médian · gris, sans produire de token ;

2i. Remplacez l'action de la règle de rattrapage par un appel à :
`error("caractère inattendu "+ yytext()).`

2j. Vérifiez que tous les tests des classes `Exo1Test` et `Exo2Test` sont désormais au vert. Si certains tests de `Exo2Test` échouent encore, lisez attentivement les messages d'erreur et corrigez votre lexer.

Exercice 3 (Littéraux de chaîne de caractères).

On souhaite maintenant reconnaître des littéraux de chaîne de caractères délimités par des guillemets doubles, et les représenter par un token spécifique.

3a. Déclarez dans l'interface `Token` un nouveau type de token `StringLiteral` pour les chaînes de caractères. Puis, dans `Main.readString()`, affichez son texte avec la couleur `FG_RED` en ajoutant une branche `case`.

3b. Pour simplifier, on commence sans séquences d'échappement : une chaîne est une suite de caractères quelconques sauf `"` et le passage à la ligne, entourée de guillemets doubles.

Proposez un motif JFlex qui reconnaît de telles chaînes et ajoutez une règle dans `lexer.flex` qui retourne un `Token.StringLiteral` avec le texte approprié.

3c. Adaptez le motif pour autoriser la séquence de caractères `\"` à l'intérieur d'une chaîne.

Indice : un littéral de chaîne de caractères est composé d'un caractère `"`, suivi d'une suite éventuellement vide de :

— soit n'importe quel caractère, sauf un passage à la ligne, `"` ou `\` (`[^"\\]`),

— soit le caractère `\` suivi de n'importe quel autre caractère, sauf un passage à la ligne (`\\.`),

suivie d'un caractère `"`