Analyse lexicale avec JFlex - 2ème partie

1 Expression arithmétique postfixée

Dans une expression arithmétique **en notation postixée**, les opérandes sont écrites avant l'opérateur. Par exemple la valeur asociée à $\boxed{3\ 5\ +}$ est 8 (équivaut à la notation infixée classique : $3 + 5$).
Pour évaluer une expression postfixée, on procède de gauche à droite en remplaçant au fur et à mesure chaque opérateur par le résultat de son évaluation. En procédant de cette manière, chaque opérateur s'applique aux 2 valeurs qui le précèdent. En notation postfixée, il n'y a pas de notion de priorité.
Exemples :

- $\boxed{3\ 5\ +\ 2\ *} \rightarrow \boxed{8\ 2\ *} \rightarrow \boxed{16}$
- $\boxed{3\ 5\ 2\ +\ *} \rightarrow \boxed{3\ 7\ *} \rightarrow \boxed{21}$
- $\boxed{3\ 5\ 2\ *\ +} \rightarrow \boxed{3\ 10\ +} \rightarrow \boxed{13}$

Le nombre d'opérandes requis par un opérateur est fixe. Ce nombre est appelé **arité**. L'arité de $\boxed{+}$ est 2. Idem pour celle de $\boxed{*}$
Une expression postfixée peut être incorrecte : soit par manque d'opérandes, soit par excès. À la fin de l'évaluation il devrait rester une et une seule valeur, et aucun opérateur.

- $\boxed{4\ 3\ 5\ +\ 2\ *} \rightarrow \boxed{4\ 8\ 2\ *} \rightarrow \boxed{4\ 16}$ expression incorrecte (trop d'opérandes).
- $\boxed{5\ 2\ +\ *} \rightarrow \boxed{7\ *}$ expression incorrecte (manque une opérande).

D'une manière algorithmique, l'évaluation utilise une **pile de nombres**. Initialement la pile est vide puis, en lisant l'expression de gauche à droite :

- à la lecture d'une valeur : l'ajouter sur la pile.
- à la lecture d'un opérateur : enlever de la pile les opérandes nécessaires (selon l'arité de l'opérateur), effectuer l'opération et mettre le résultat sur la pile.
  NB : si la pile ne contient pas le nombre d'opérandes nécessaires, c'est que l'expression est incorrecte.

À la fin, la pile ne devrait contenir qu'une seule valeur (le résultat de l'expression complète).

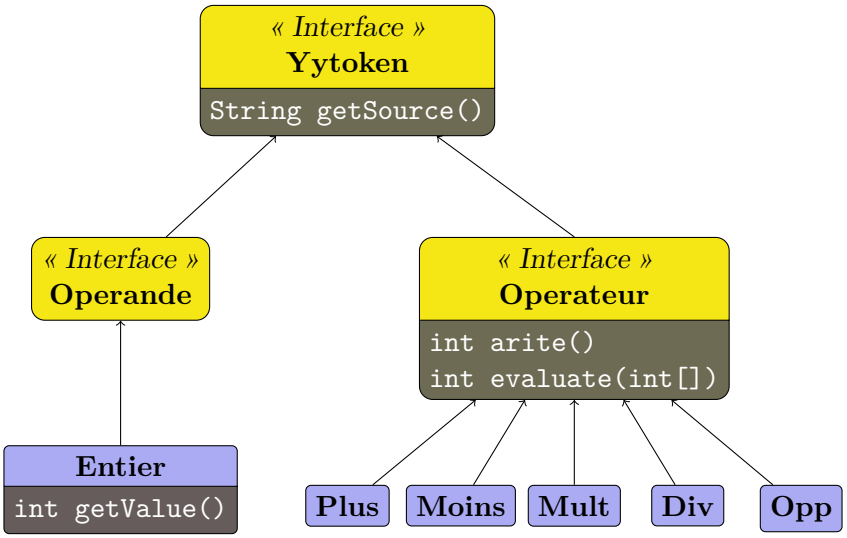
1.1 Les tokens à prendre en compte

Dans cette première version, on considèrera les tokens suivants :

- Entier (sans signe) : suite non vide de chiffres décimaux ne commençant pas par 0 ou suite non vide de chiffres en octal commençant par 0, ou suite non vide de chiffres en hexadécimal **précédée** de `0x` ou de `0X` (pour l'ensemble de l'hexadécimal on admet les majuscules et les minuscules)
- opérateurs : `+ - * /` (arité 2) et le mot clé `opp` ou `OPP` de arité 1 (opposé).

1.2 L'implémentation des tokens

Une hiérarchie de types (interfaces, classes abstraites, classes) vous est fournie pour implémenter les tokens. Contrairement à l'exercice précédent, les tokens sont implémentés par des classes différentes qui reflètent leur typologie. Chaque classe implémente `Yytoken`(qui est donc une `interface` et non plus une classe) .
De cette manière, on peut associer au token des méthodes propres à sa sémantique. Par exemple la classe `Entier` implémente une méthode permettant d'obtenir sa valeur et chaque classe d'opérateur implémente la méthode d'évaluation.



(note : les classes abstraites ne sont pas représentées sur ce diagramme)

Exercice 1 :
Vous trouverez l'ensemble des fichiers java dans le dossier `postfixee`
Deux classes "principales" sont présentes :

- une classe (avec main) `Decoupe`, utile à la mise au point de l'analyseur lexical (test du découpage en tokens).
- une classe (avec main) `Evaluateur` qui implémente l'évaluation de l'expression.

(NB : les classes `Ident` et `Stockage` ne seront utiles qu'à l'exercice suivant)

Écrivez le fichier `.lex` nécessaire pour terminer l'implémentation de l'évaluateur d'expressions postfixées. Vous autoriserez la présence de commentaires (reprenez les commentaires de type 1 et 3 de l'exercice de la fiche précédente).
Il vous est **très fortement** conseillé de tester d'abord le bon fonctionnnement de l'analyseur lexical avec l'appli `Decoupe`.

2 Les états de l'analyseur lexical

L'analyseur implémente la notion d'**état**. À tout instant l'analyseur se trouve associé à un état.
Il existe un unique état prédéfini qui s'appelle `YYINITIAL` et l'analyseur est donc, par défaut, en permanence dans cet état.

- la création d'états supplémentaires passe par la directive `%state` suivie de la liste des noms d'états à créer (dans la partie 2 du fichier .lex).

  ```
  %state ETAT_1, ETAT_2
  ```

- une règle (partie 3 du fichier .lex) peut être rendue conditionnelle, en la préfixant par une liste de noms d'états figurant entre `<  >`. Par exemple la règle :

  ```
  <YYINITIAL,ETAT_1> [a-z]+ { return ...; }
  ```

  ne s'applique **que** si l'analyseur est dans l'état `YYINITIAL` ou dans l'état `ETAT_1`.
- on peut regrouper des règles soumises à une même condition

  ```
  <YYINITIAL,ETAT_1>  {
      /* toutes les règles figurant ici seront soumises à la condition */
  }
  ```

- un changement d'état est commandé par l'instruction JAVA `yybegin(`*nom_d_etat*`);` qui figure généralement dans la partie « action » d'une règle.

  ```
  <YYINITIAL,ETAT_1>  {
      [a-z]+  { yybegin(ETAT_2); return ...; }
  }
  ```

2.1 Un exemple

Un exemple vous est fourni dans le dossier `exemple_etats`. Lire le fichier `.lex` pour les explications.

Exercice 2 :

L'évaluateur d'expression postfixée dispose maintenant de variables (de type entier, évidemment). Une variable est dénotée par un identificateur qui suit la même syntaxe que dans la fiche précédente. Dans un nom de variable, majuscules et minuscules ne sont pas équivalentes (`a` et `A` désignent des variables différentes). Les variables ne sont pas déclarées. Une variable non initialisée possède la valeur 0.

Pour désigner une valeur, on peut maintenant utiliser un nom de variable de la même façon qu'on utilise une valeur immédiate. Par exemple `4 toto +` vaudra 4 si *toto* n'a pas été initialisée (et vaut donc 0).

La commande `->` suivie d'un nom de variable signifie qu'il faut copier la valeur située au sommet de la pile dans la variable (nb : la valeur n'est pas enlevée de la pile). La commande ne doit donc pas être utilisée quand la pile est vide. Par exemple :

- `3 5 + -> a 4 * a *` vaut 256 (a prend la valeur 8)
- `0x10 -> x x *` vaut 256 (x prend la valeur 16)

On peut trouver un ou plusieurs espaces ou fins de ligne entre le `->` et l'identificateur qui suit. Par contre on ne peut **rien** trouver d'autre (pas de commentaires).

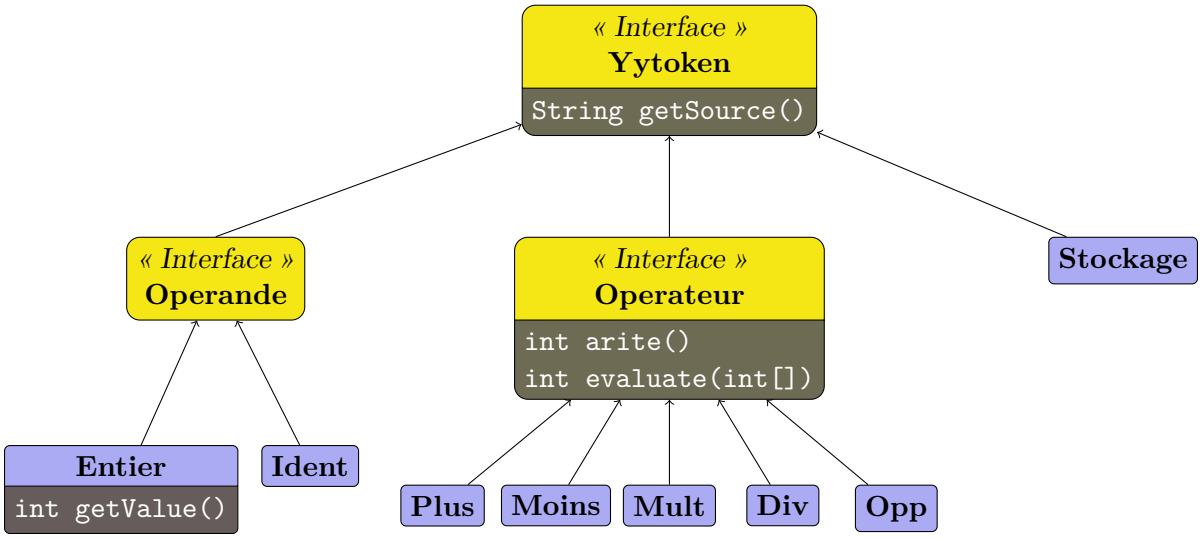
Deux nouvelles classes Java : `Ident` et `Stockage` vous sont fournies. `Ident` implémente le token identificateur quand celui-ci est utilisé simplement (non précédé de `->`).

`Stockage` implémente le token identificateur quand celui-ci est précédé de `->`. Pour ce token la valeur « source » est le source de l'identificateur seul (sans le `->` ni les espaces éventuels)

`->` n'est pas considéré comme un token.

Par exemple : `0x10 -> x x *` →

Entier	Stockage	Ident	Mult
`0x10`	`x`	`x`	`*`



Implémentez cette nouvelle version de l'évaluateur d'expressions (il vous reste à construire le fichier `.lex`, en utilisant les états).