Compilation
TP Lexeur/Parseur
Java avec JavaCC

15 février 2021

Prérequis : ce TP suppose que vous avez initialisé un dépôt *git* avec un commit initial (avec uniquement le *readme*). Si ce n'est pas le cas, faites le.
Au début du TP, vous devez vous placer dans le dossier cloné, vous serez automatiquement sur la branche `master`, qui sera la branche de rendu pour la seconde partie compilateur (si vous préférez qu'elle ai un autre nom, vous pouvez le changer).

A la diférence des générateurs de parseurs utilisés dans les versions alternatives de ce TP, javaCC intègre son propre générateur de lexeur lexeur et utilise un algorithme LL. Le premier point est généralement bien pratique, tandit que le second est assez restrictifs sur certains aspects de la grammaire, mais ne devrait pas être un obstacle pour votre projet. [1]

Pour ce projet, contrairement au cours de Java distribué, nous vous déconseillons d'utiliser un IDE (Eclipse, Netbean ou InteliJ). C'est simplement car toutes les manipulations externe à java (génération de parseur, utilisation de Git, récupération de code) devra être intégré correctement à votre IDE et que nous, enseignants, n'avons pas le temps de faire un tutoriel spécifique à chaque IDE. Si vous choisissez leur utilisation, c'est à vos risques et périles.

Exercice 1 (Le parseur (+lexeur)).

1. Créez un fichier `projet.jj` qui génèrera notre parseur le programme exécutable :

```
// fichier Compilateur.jj
PARSER_BEGIN(Compilateur)
import java.io.InputStream;
public class Compilateur {
    public static void main(String args[]) {
        try {
            Compilateur parseur = new Compilateur(System.in);
            parseur.mainNT();
```

1. Faites juste attention avec le contenu optionel du projet.

```
                System.out.println("C'est bien une expression arithmetique");
            } catch (TokenMgrError e) {
                System.out.println("Ceci n'est pas une expression arithmetique");
            } catch (ParseException e) {
                System.out.println("Ceci n'est pas une expression arithmetique");
            }
        }
}
PARSER_END(Compilateur)

SKIP :
{ " " | "\t" }
TOKEN :
{   < NOMBRE: ["1"-"9"] (["0"-"9"])* >
  | < EOL: "\n" >
}

void mainNT () :
{}
{ expression() <EOL> }

void expression () :
{}
{ terme() (
      "+" terme()
    | "-" terme()
    )*
}

void terme () :
{}
{ facteur() (
      "*" facteur()
    | "/" facteur()
    )*
}

void facteur () :
{}
{   "(" expression() ")"
  | "-" facteur()
  | <NOMBRE>
}
```

Dans l'ordre :

— entre `PARSER_BEGIN(Compilateur)` et `PARSER_END(Compilateur)` on

écrit les imports et les méthodes disponibles dans la classe de sortie, [2]

— dans le main, on "construit" un parseur lisant l'entrée standard avec la ligne `parseur = new Compilateur(System.in)`, on le lance avec `parseur.mainNT();`,
— le `try{...}catch{...}` récupère toutes les exceptions TokenMgrError et ParseException qui sont, respectivement, les erreurs de lexing et de parsing.
— la partie commençant par `SKIP : ...` décrit les séparateurs, pour l'instant juste les espaces et les tabulations, mais on en aura d'autre dans le future (les commentaires par exemples).
— la partie commençant par `TOKEN : ...`, déclare les noms de tokens,
— `void mainNT(): {}{..}` déclare un non-terminal `mainNT()` qui crée un `void` en lisant une expression suivit d'un passage à la ligne,
— `void expression(): {}{..}` déclare un autre non-terminal `expression()` qui créé un `void` en lisant un terme suivit d'une séquence de sommes et soustractions de termes,
— idem pour les autres non-terminaux.
— la grammaire est compliquée pour plusieurs raisons (on verra comment la simplifier ensuite) :
  — Les non-terminaux `expression`, `terme`, et `facteur` permettent d'encoder la priorité : on fat d'abord toutes les opérations additives dans `expression`, puis, sous ces opérations additives, on peut faire les opérations multiplicatives de `terme` (puis on peut avoir des atomes ou bien le moins unitaire qui a la priorité maximal).
  — La structure de `expression` utilise version condensée de la grammaire, avec des notations d'expression régulière. En effet, il faut le lire comme "une `expression` est un `terme` suivi de zero, une, ou plusieurs séquences de la forme `"+" terme` ou verb|`"*"` terme|".

2. Compilez ça à l'aide des commandes suivantes dans le terminal (ou faites un *makefile*) :

```
$ javacc Compilateur.jj
$ javac Compilateur.java
```

Cela génère plein de classes annexes intermédiaires, parmi lesquels votre lexeur `JavaParserTokenManager.java`, et votre parseur `Compilateur.java`. Si la commande `javacc` ne passe pas, installez l'application (paquet de même nom sur linux, sinon c'est un programme java que l'on peut trouver sur internet).

3. Vous pouvez lancer `java Compilateur` dans un terminal et taper une expression avant d'aller à la ligne. Essayez-en quelques-une pour voir les diférents comportements.

4. Vous pouvez archiver ce fichier au suivit de git, commit-er et push-er :

```
$ git add Compilateur.jj
```

2. Pour l'instant et par simplicité on met le main ici.

```
$ git commit -m "mon premier parseur"
$ git push
```

Remarquez que je n'ajoute pas les fichiers générés, même pas les `.java`. On n'archive jamais les fichiers générés car ceux-ci sont trop changeant, le "dif" représentant alors tout le fichier, et l'archivage devenant quadratique en espace.

5. On va introduire une *feature*, pour ne pas polluer la branche master, on crée une branche séparée (ici appelée `parser_work`) et on rentre dedans :

```
$ git checkout -b parser_work
```

6. Jusque là, on lançait le scan lorsque l'on arrivait sur une nouvelle ligne (lorsque l'on voit le token `EOL`). On voudrait pouvoir écrire plusieurs lignes et finir par un " ;". Pour ça ajoutez le retour à la ligne `'\n'` parmi les séparateurs, enlevez le token `EOL`, et remplacez son utilisation dans la grammaire par " ;" afin de stoper l'évaluation à ce moment.
7. Faites un commit et un push de vos changements :

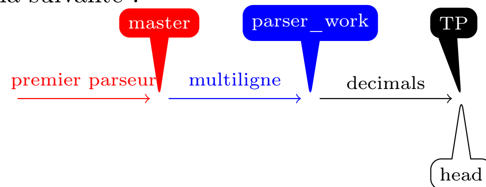
```
$ git add Compilateur.jj
$ git commit -m "multilignes"
$ git push
```

Remarquez que l'on refait un `git add`, en fait celui-ci sert aussi à spécifier les fichiers que l'on a mis à jours.

8. On veut maintenant voir si on peut rajouter une feature du fragment 0.1, or on n'a pas encore tout ce qu'il faut pour le fragment 0.0, on peut tout de même s'avancer en créant une nouvelle branche temporaire :

```
$ git checkout -b TP
```

9. Rajoutez une écriture décimal (virgule fixe) pour nos nombres. Pour ça, il vous faut modifier l'expression régulière les reconnaissant.
10. Faites un nouveau commit. La situation de votre dépôt devrait alors être la suivante :



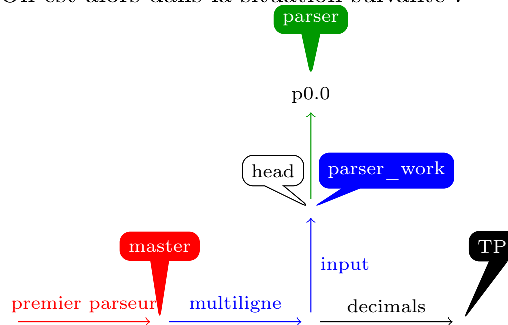
Exercice 2 (Simplification et ouverture de fichiers).

1. Revenez sur la branche `parser_work`.

2. On voudrait maintenant finir le fragment 0.0 du parseur, mais pour ça, il faut permettre à l'utilisateur de donner un fichier en argument. Pour ne pas casser ce que l'on a déjà, on va lancer la version terminal lorsque l'utilisateur ne donne pas d'argument, sinon on ouvre le premier argument comme un fichier. Implémentez cette feature.
3. Faites un commit, puis créez la branche de rendue `parseur` et tagg-ez le commit et revenez sur la branche de travail :

```
$ git add main.ml
$ git commit -m "input"
$ git checkout -b ``parser''
$ git tag -a p0.0 -m ``premiere version faites en TP par <mon nom>''
$ git push
$ git checkout ``parser_work''
```

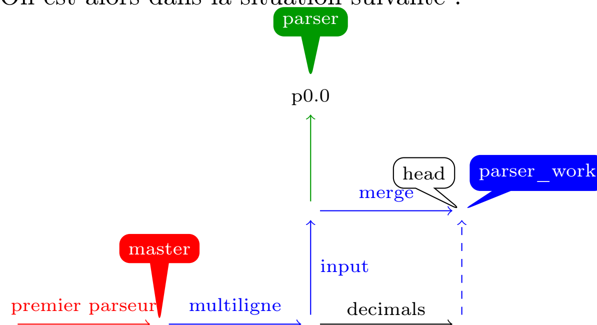
   On est alors dans la situation suivante :



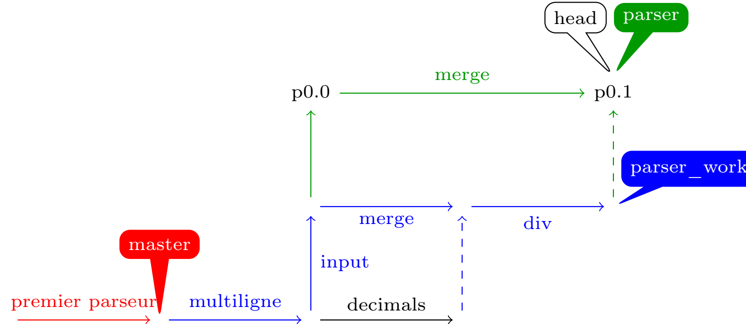
4. On va maintenant pouvoir récupérer les décimaux que l'on avaient ajouter tout à l'heure. Pour ça il suffit de merger avec `TP`, après quoi on peut supprimer `TP` qui n'est plus utile :

```
$ git merge TP
$ git push
$ git branch -d TP
$ git push --delete TP
```

   On est alors dans la situation suivante :



5. Rajoutez l'opérateur de division `(_/_)` (attention à sa priorité !), faites un commit, revenez sur la branche `parser` et placez le *tag* `p0.1` :



Exercice 3 (Aparte : interpréteur). Il n'est pas demandé, dans le projet, de faire un interpreteur car ce serait plus difficile qu'un compilateur vers notre assembleur abstrait. Mais pour les tout premiers fragments, c'est plus simples, et on va le faire pour se familiariser avec l'outil de parsing.

1. Revenez sur master, créez une nouvelle branche appelée `interpreter`.
2. Commencez par modifier la valeur de retour des non-terminaux en remplaçant `void` par `int`. Celà signifie qu'en découvrant l'arbre syntaxique, plutôt que d'oublier ce qu'il a lu, le parseur calculera un entier à chaque noeud qui est la valeur issu du calcul du sous-arbre correspondant.
3. Remplacez l'avant dernière ligne par :

   ```
     | t=<NOMBRE>                {return Integer.parseInt(t.image);}
   ```

   Celà récupère le lexem t associé au nombre, le décode comme un entier et l'associe à ce noeud de l'arbre syntaxique.
4. Modifiez la ligne du dessus afin de récupérer l'entier calculé par le facteur e et de renvoyer, à ce noeud de l'arbre syntaxique, sa négation $-e$:

   ```
     | "-" (e=facteur())         {return (-e);}
   ```

5. Faites pareil avec le noeud des parenthèses.
6. Attention, en java, il faut toujours déclarer ses variables, en JavaCC aussi. Les variables e et t se déclarent dans l'accolade juste après la déclaration de `facteur` :

   ```
   int facteur () :
   { int e;
     Token t;
   }
   {
       "(" (e=expression()) ")"  {return e;}
     | "-" (e=facteur())         {return -e;}
     | t=<NOMBRE>                {return Integer.parseInt(t.image);}
   }
   ```

7. Pour les termes et les expressions, le code et un petit peu plus subtile, le voici pour les termes :

```
int terme () :
{ int res;
  int e;
}
{
    e=facteur()                 {res = e;}
    ( "*" (e=facteur())         {res = res * e;}
    )*                          {return res;}
}
```

   Ici, il faut voir un noeud de `term` comme une liste de multiplications, on va parcourir cette liste en accumulant les opérations une à une dans la variable `res` que l'on retourne à la fin. Notez qu'en procédant ainsi on applique implicitement une associativité gauche aux opérations de multiplication. Ça tombe bien car c'est celle qu'il faut ! Si on avait eu besoin d'une asociativité droite ça aurait été beaucoup plus dificile.

8. Faites de même avec les expressions.

9. N'oubliez-pas `MainNT` :

```
int mainNT () :
{int e;}
{ (e=expression()) ";"  {return e;}}
```

10. Il ne vous reste plus qu'a modifier le main :

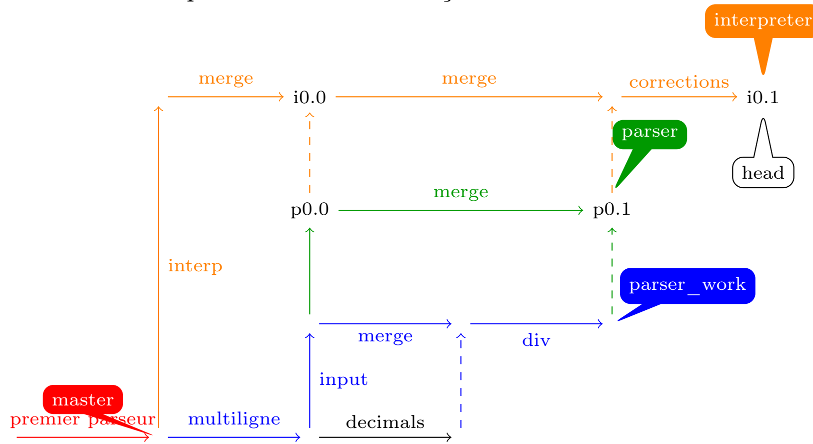
```
    public static void main(String args[]) {
        try {
            Compilateur parseur = new Compilateur(System.in);
            int i = parseur.mainNT();
            System.out.println(i);
        } catch (TokenMgrError e) {
            System.out.println("Ceci n'est pas une expression arithmetique");
        } catch (ParseException e) {
            System.out.println("Ceci n'est pas une expression arithmetique");
    }   }
```

11. Vous pouvez tester et faire un commit (appelé `interp` dans la suite) si tout va bien.

12. Faites le merge avec `p0.0` afin de pouvoir prendre des fichiers en entrée. Corrigez les conflits si besoin et tagg-ez la version `i0.0`.

13. Faites le merge avec `p0.1`. Le programme résultant ne sera pas fonctionnel car on a typé nos sorties avec des entiers au lieux de flottants et car la division n'est pas écrit. Corrigez tout ça, faites un commit et tagg-ez le.

Votre dépôt doit ressembler à ça :



Dans les exercices suivants, on oubliera la branche “interpreter”. En effet, l’interpréteur n’est pas demandé pour le projet contrairement au compilateur (l’interpréteur devient difficile une fois que l’on parle de variable, très difficile lorsque l’on introduit les fonctions, et encore plus avec les exceptions).

Exercice 4 (Créer un AST en sortie). Avant de pouvoir faire un interpreteur, on veut pouvoir manipuler notre AST. Ce n’est pas strictement nécessaire au début, mais c’est très pratique pour s’y retrouver et séparer les problèmes, et ça deviendra essenciel lorsque le fragment de langage grossira.

Attention, il y a deux encodages possible pour faire des arbres d’arrité variable (comme les ASTs) en Java :

— La première est d’utiliser des arbres dont les noeuds sont des listes d’objets que l’on caste dans tous les sens. Celà revient à ignoré le système de type statique. C’est un peu plus simple tant que l’on n’est pas trop gros, mais les erreurs arrivent vite et on ne s’en apperçoit qu’a l’exécution. Ce n’est pas la voie choisie, mais des outils (comme JJTree par exemple) utilisent ce moyen.

— La seconde est d’utiliser, pour chaque non terminal, une classe abstraite dont ses concrétisations sont les diférents types de noeuds associés. C’est ce que cons avons choisi d’utiliser, d’où le grand nombre de classes.

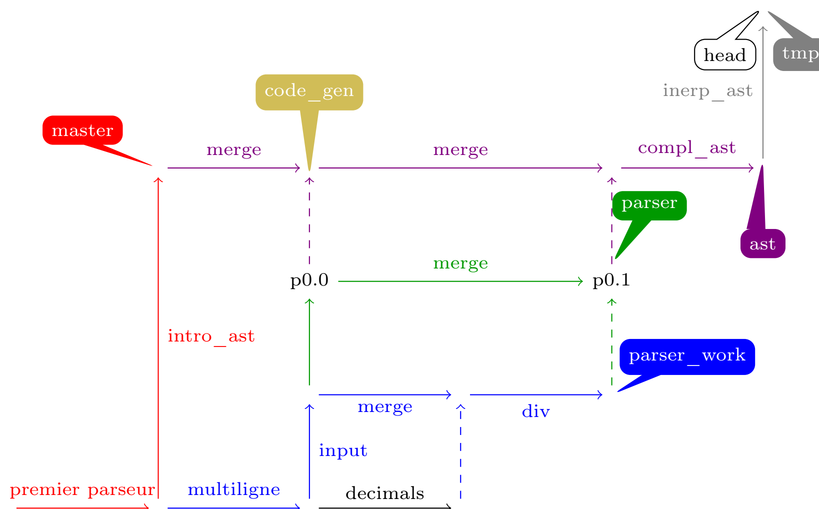
1. Revenez sur Master, et téléchargez le package d’AST disponible ici. Celui-ci doit être intégré comme un pakage séparé de votre projet afin de ne pas vous perdre dans les multiples classes, nous vous invitons aussi a intégrer le compilateur généré comme un Package.

2. Vous y trouverez plein de classes, essayez de dessiner la hierarchie de classes pour y voire un peu plus clair :

   — La classe abstraite `AST` n’est pas très utile, on la rajoute au cas où on en aurait besoin. L’idée est de la sous-classer par chaque sorte de l’AST.

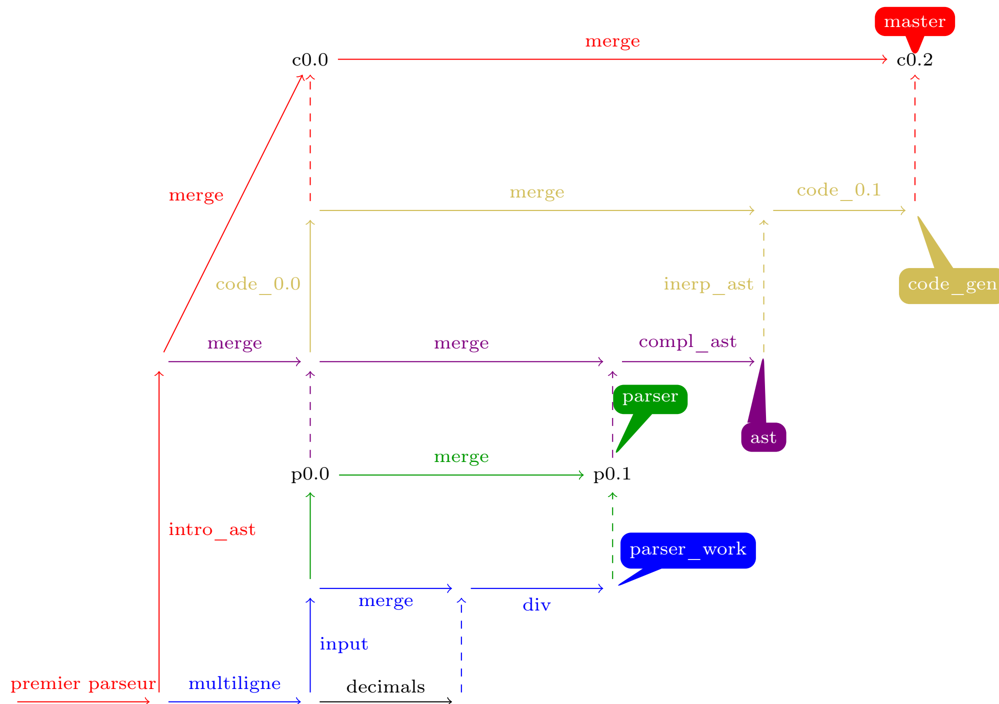
   — La classe abstraite `ExpressionA` est la classe des AST d'expression. Pour l'instant elle est vide car la seule méthode que l'on utilise est `toString` qui est toujours présente, mais pour chaque méthode que l'on va ajouter sur nos arbres, il va faloir ajouter sa signature.
   — Les classes `Neg, Num, Plus, Moins` et `Mult` sont les concrétisations de `ExpressionA`, ce sont les diférents types de noeuds de notre AST.
   — La classe abstraite `ExpressionA_Binaire` permet simplement de factoriser le code des expressions binaires, qui se comportent souvent de la même manière.

3. Modifiez `Compilateur.jj` affin d'utiliser ce packedge :
   — Changez le type de retour des non-terminaux affin de retourner un `ExpressionA`,
   — Changez le main, comme pour l'interpreteur, affin d'afficher l'AST calculé.
   — Changez le code de chaque terminaux comme pour le code de l'interpreteur, mais de manière à appeller un constructeur concret de noeud à chaque action.
4. Vous pouvez tester et faire un commit (appelé `intro_ast` dans la suite) si tout va bien.
5. Faites une nouvelle branche `ast` et mergez avec `p0.0`.
6. Ceci n'est pas encore notre compilateur, il faut encore générer du code, mais avant ça, nous allons nous entraîner à maîtriser l'AST. Pour pouvoir revenir sur la génération de code, veuillez créer une nouvelle branche `code_gen` puis revenir sur `ast` et mergez avec `p0.1`.
7. Rajoutez l'AST de la division et des flottants.
8. Sur une branche temporaire séparée (ceci est un exercice, et n'est pas utile pour le projet), écrivez un programme qui évalue l'expression entrée en parcourant l'arbre syntaxique généré.



Exercice 5 (Génération de code).

1. Revenez sur `code_gen`,
2. Dans `ExpressionA.java`, ajoutez une méthode abstraite `public String toAssembly()` dont les concréisations vont permetre de parcourir l'arbre et de faire un affichage post-fixe de sa structure, ceci en affichant :
   — la ligne "`CsteNb` n" après avoir quitté un noeud `Nombre(n)`,
   — la ligne "`AddiNb`" après avoir quitté un noeud `Plus`,
   — la ligne "`MultNb`" après avoir quitté un noeud `Mult`,
   — la ligne "`SubiNb`" après avoir quitté un noeud `Moins`,
   — la ligne "`NegaNb`" après avoir quitté un noeud `Neg`,
3. Dans le cas où fichier est donné en argument de notre compilateur, on voudrait que soit généré automatiquement un nouveau fichier de même nom mais avec l'extension `.jsm`
4. Testez, commitez, mergez `master` avec cette branche, placez votre tag `p0.0`, et revenez sur la branche `code_gen`.
5. Mergez avec la branche `ast`, complétez la génération de code, testez, commitez, mergez `master` avec cette branche, puis placez votre tag `p0.1`.



Vous pouvez maintenant passer au TP2. À partir de maintenant les TPs :
— seront langage agnostiques,
— ne donneront plus de détails d'implémentation, et
— ne préciseront plus la structure interne de votre dépôt git.

Seules les brancher `master` et `parseur` sont demandées, pour les autres, vous pouvez fonctionner comme bon vous semble. Mais il est fortement recommandé de fonctionner ainsi par couches successives, c'est plus pratique pour débuguer, mais aussi pour travailler en parallèle avec des vitesses différentes. N'hésitez pas non plus à poser d'autres tags afin de pouvoir merger un point particulier tout en continuant sur une branche.