Compilation
TP Lexer and Parser
in Java, with JavaCC

Flavien Breuvart

janvier 2025

Prerequits : This tutorial is expecting you to have completed TP0, or at least to have initialised a *git* repository with an initial commit (with a readme and eventually an hello_word and a makefile) and 5 branches (`parser_work`, `parser`, `ast`, `code_gen` and `main`).
At the start, you are also asumed to have a terminal working on the main directory of your project (the one with the `.git`) and on the branch `parser\_work`.

Contrary to parser generators in the alternative C and Ocaml version of this tutorial, *JavaCC* contains its own leser generator; in addition it ses a LL* algorithm which is not from the LR familly seen in courses. The first happen to be quitte usefull, but not the second as LL algorithms are more restrictive and you will have to contortionate the grammar to get expected priorities.[1]

Exercice 1 : The parser (+lexer)

1. Verify that `javacc` is installed on your computer by tryin the command-line. If it is not the case, follow the instruction to get alocal version, or une the eclipse plugin, or any other version you may find online (`javacc` is just a java proram).
   EDIT: On the universiti's computers, get javaCC archive (or the git version), extract it on `~/javacc/`, rename the subrepertory `~/javacc/bootstrap` into `~/javacc/target`, and add, in your `bashaliases` the path extention to the scripts :

   ```
   export PATH=''$PATH:~/javacc/scripts/''
   ```

2. Create a file `projet.jj` that generates your parser as an executable program :

   ```
   // fichier Compilor.jj
   PARSER_BEGIN(Compilor)
   import java.io.InputStream;
   public class Compilor {
       public static void main(String args[]) {
           try {
               Compilor parser = new Compilor(System.in);
               parser.mainNT();
               System.out.println("This is an arithmetic expresion");
           } catch (TokenMgrError e) {
               System.out.println("This is not an arithmetic expresion");
           } catch (ParseException e) {
               System.out.println("This is not an arithmetic expresion");
           }
   ```

[1] Be especially carefull for optional content.

```
    }
}
PARSER_END(Compilor)

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

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

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

void term () :
{}
{ factor() (
      "*" factor()
    )*
}

void factor () :
{}
{   "(" expression() ")"
  | "-" factor()
  | <NUMBER>
}
```

Sequencially:

- between `PARSER_BEGIN(Compilor)` and `PARSER_END(Compilor)`, are wrotten the imports and methodes required for the generated class,[2]
- the *main* methode is constructing a parser `parser = new Compilor(System.in)` that reads the standard input with the line, then it runs this parser with `parser.mainNT();`,
- the `try{...}catch{...}` is collecting exceptions `TokenMgrError` and `ParseException` that are, respectivelly, lexing error and parsing error.
- the section starting with `SKIP : ...` is a description of separators, for now, those are just spaces and tabulations, but you will latter add other (new lines and comments).
- the section starting with `TOKEN : ...`, is the declaration of the tokens,

[2]For now, and for simplicity, the main is written there, but you can change it position later on.

- `void mainNT(): {}{..}` declares a terminal `mainNT()` recognises an expression followed by a new line ("EOL" stands for end of line),
- `void expression(): {}{..}` declares another non-terminal `expression()` that reconises term followed by a sequence of sums and substractions of terms,
- idem for the two newt non-terminals.
- the grammar seem quitte complex, here are the reasons :
  - The non-terminals `expression`, `term`, and `factor` are all representing expressions, but force the priorities between operations : additive operations, which are of lesser priorities, are done in `expression` which mean above the other operations, bellow are the multiplicative operations, in `term` and bellow are factors that are potential unary negations of numbers of other expressions guarded by parentheses.
  - The structure of `expression` uses a condensed version of the grammar, with regexp-like notations. Indeed, it should be read as "an `expression` is a`term` followed by zero, one, or more sequences of the form `"+" term` or of the form verb|`"*"` term|".

3. Compile it using the following command lines (later, you can try to write a*makefile*):

   ```
   $ javacc Compilor.jj
   ```

   This generates a lot of intermediate classes, among which the lexer `JavaParserTokenManager.java`, and the parser `Compilor.java`. If the command `javacc` do not work and you have the program locally, you can run it :

   ```
   $ java javacc Compilor.jj
   ```

   Then you can compile the main program

   ```
   $ javac Compilor.java
   ```

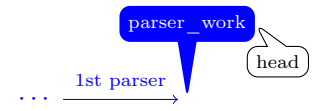
4. Run it with `java Compilor`, you can now tap an expression which will be tested each time you write a new line :

   ```
   $ java Compilor
    1+1+-(42*2)
    This is an arithmetic expresion
   $ java Compilor
    1+1++(42*2)
    This is not an arithmetic expresion
   ```

5. Archivate this file on the it repository :

   ```
   $ git add Compilor.jj
   $ git commit -m "1st parser"
   $ git push
   ```

   Important : never add the generated files, not even those of the `.java` extension. If you do, you will get tons of conflicts and weird behaviors.

   Here is the expected historic (without the part comming from TP0) 

6. Add a new terminal, that of `command`, which are expressions followed by a ";". Proceed by creating a new terminal, which will call for an `expression` and will be called by `mainNT` (replacing the call to `expression`).

7. Commit and push the changes :

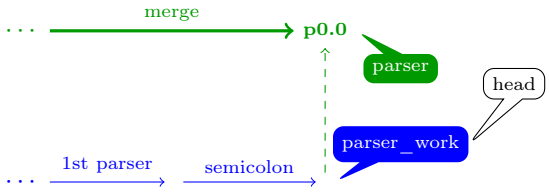
```
$ git add Compilor.jj
$ git commit -m "semicolon"
$ git push
```

--- * ---

Exercice 2 : Fragment `0.1`, parser

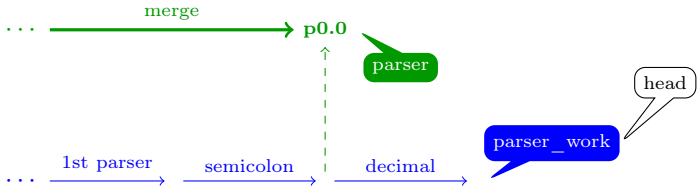
1. Commit, and sitch to `parser` branch, merge the `parser\_work` branch, ad *tag* the version before comming back to the working branch :

```
$ git switch parser
$ git merge parser_work
$ git tag -a p0.0 -m ``first version done using the tutorial by <my name>''
$ git push --tags
$ git checkout parser_work
```

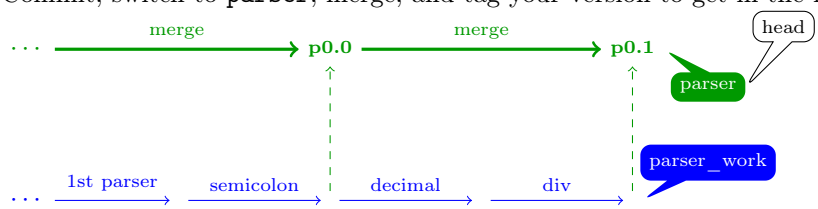


Remark: remember to *push* using the option `-tags`, otherwise, your tags will not be registered in the repository.

2. Modify the regular expression of `NUMBER` for it to accept decimal numbers (foats without scientific notations). Remember that in javascript, there are no integer, numbers without comma are still foats. After a commit you should have



3. Add a division opperator `/`. Try to get the correct priority (even though you can't test it yet). Commit, switch to `parser`, merge, and tag your version to get in the following situation :



--- * ---

Exercice 3 : Aparte : interpreter

It is not required, in the project, to write a full interpreter which would be more complex to do than the actual project due to the need to correctly evaluate variables and functions. Nonetheless, for our very small fragment without variable, it is easy and a good exercice.

1. We will use a new gi branch called `interpreter`, which start from `p0.0` :

```
$ git branch interpreter p0.0
$ git switch interpreter
```

2. Start by modifying the return type of non-terminals, replacing `void` par `int` in each "function". This mean that while exploring the syntactic tree, rather than forgeting each red node, we will compute the integer resulting from the comutation.

3. Replace the antepenultiem line by:

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

   This retrieve the *Stringt* associated to the token `NUMBER`, decodes it as an it (our small lexer is givin a string), and mapping it to the result of the current node.

4. Modify the line above in order to retrieve the computed factor *inte* and returning, for this node of the syntactic tree, the negation $-e$:

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

5. Proceed similarly with the last case.

6. Be carefull, in javacc, as for java, you need to declare each used variable. Thus variablesL e and t have to be declared, the acolades just after the function declaration are here for you to write those declarations (anything you write here will be copied at the begining of the generated function) :

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

7. For terms and expressions, the code is slightly more subtle:

```
int term () :
{ int res;
  int e;
}
{
    e=factor()                  {res = e;}
    ( "*" (e=factor())          {res = res * e;}
    )*                          {return res;}
}
```

   Here, a node `term` is a list of multiplications, we will traverse is while accumulating operations in the variable `res` which we returns at the end of the loop. Notice that this implicitly applies left assiciativity to operands (while recursive call applis right associativity).

8. Completes the last non-terminal similarly.

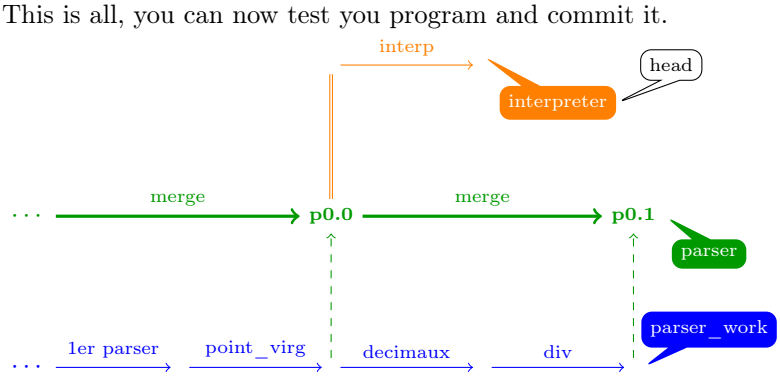
9. Do not forget `MainNT` :

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

10. Modify the `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");
    }   }
```

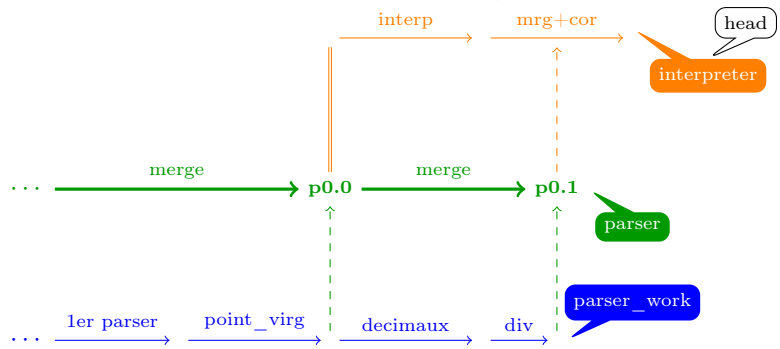
11. This is all, you can now test you program and commit it.



12. Merge `p0.1` in the interpreter.[3] There should be conflicts due to the division. Correcting this merge is non-trivial as you really have to mix both lines by hand. Do it. (Recall that after

[3]Remark : you could have equivalently merge with `parser` which is curently pointing to `p0.1`.

correcting a conflicting merge you have to `git add` the conflicting files and `commit` the changes).



13. Try your new version, and test it. Surprise : the result do not compile.
    Warning: the comited version is incorrect. We are not using a *realease* branch but a *working* one, this is thus perfectly expected, especially after a merge.
    The issues are 1. everything is typed with int in place of floats (or double) and 2. division is not maped to any action. Correct those issues and commit.

In the following exercises, the `interpreter` branch will not be used anymore. Indeed, we wand to write a compiler, not an interpreter.

--- * ---

Exercice 4 : Creating and returning an AST

Before writing a compiler, we need to be able to create an AST. This is not strictly necessary at the begining (one could just returns the assembly directly) but the larger our fragment will become, the more dificult it would be, while spliting our problem is two (that of creating an AST and that of generating assembly) will eventually smoothen the dificulties.

Disclamer; there are two variation on the way an heterogeneous tree can be encoded in Java :

- The first uses tree which nodes are lists of type `Object`, and then cast these `Object o` whenever we use them. This basically mean that we et ride of the types which can be seen as hindrances in an heterogeneous tree, but this is obviously error prone. This is not the variation choosen for this tutorial, but `javacc` uses those behind your back via the libraty `JTree`; thus most tutorials on line will follow this part.
- The second use, for each (meaningfull) non terminal, an abstract class which concretisations correspond to its different rules, which is very heavy at the begining, but not so after some time.

1. Switch th the `ast` branch and mere the *tag* `p0.0`.
2. Download the AST package avalable here. This sould be integrated as a *package* diferent from that of your project in order to manae your multiple classes. If you do not know/want anything to do with java packages, just errase the first line in each of the files (the one with "`package AST;`").
3. You will find many classes, try to write a class diagram to understand their hierarchy:
   - THe abstract class `AST` should not be used, it is added just because we may be able to factorise a bit later on. The idea is to have a sub-class for iach kind of AST.
   - The abstract class `ExpressionA` is that of AST which represents expressions. For now it is empty thince the only methode we use is `toString` which is always implemented, but each methodes that we will add will corresponds to an abstract methode in `ExpressionA`.

- The classes `Neg, Num, Plus, Moins` and `Mult` are concretizations of `ExpressionA`,those are the diferent notes of our AST.
- The abstract class `ExpressionA_Binaire+` is only used to factorise the code of binary expressions that often behave similarly.

4. Modify `Compilateur.jj` in order to use this package:

   - Change the return types of non-terminals for that of `ExpressionA`,
   - Add actions in each terminals following the idea of the interpreter. Here is the example for terms

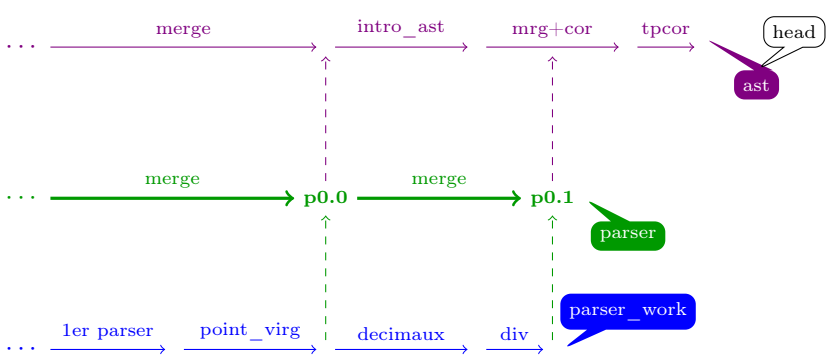
```
ExpressionA term () :
{  ExpressionA res;
   ExpressionA e;
}
{
    e=factor()                  {res = e;}
    ( "*" (e=factor())          {res = new Mult (res,e);}
    )*                          {return res;}
}
```

   - Change the `main`:

```
public static void main(String args[]) {
    try {
        Compilateur parseur = new Compilateur(System.in);
        ExpressionA 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");
}   }
```

5. You can now test it and commit commit.

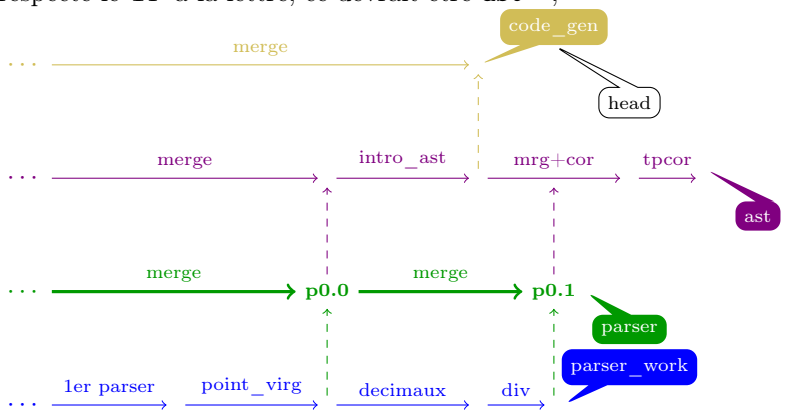
6. Merge with `p0.1`. As in the last exrcice you will have to deal with float, but you also have to deal with commands, that have their own abstract class `CommandA` similar to that of `ExpressionA`, but with only one concretisation `Expr_Semi_Col` which constructor take an `ExpressionA`.



--- * ---

Exercice 5 : Génération de code

1. Revenez sur `code_gen` et *mergez*-le avec la version de `ast` issue de `intro_ast`, si vous avez respecté le TP à la lettre, ce devrait être `ast^^`,



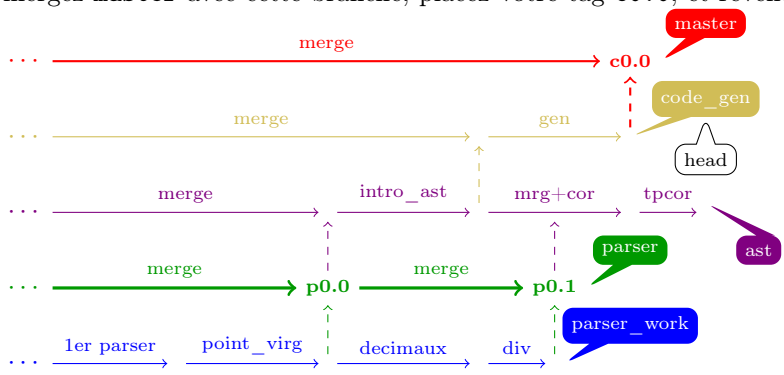
2. Dans `ExpressionA.java`, ajoutez une méthode abstraite `public String toAssembly()` dont les concréisations vont permetre de parcourir l'arbre et de faire un affichage post-fixe de sa structure, ceci en affichant :

   - 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`. Le cas où on ne donne pas d'argument n'est pas spécifié, vous pouvez faire comme vous voulez.

4. Testez votre code à l'aide de la miniJSMachine.

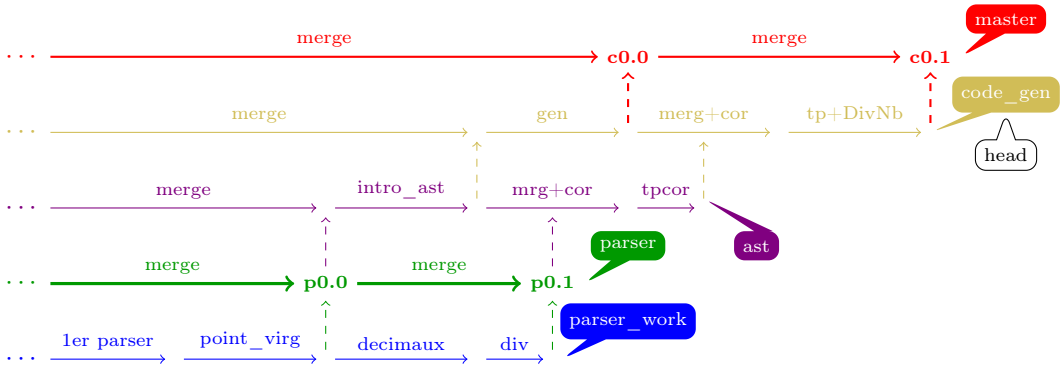
5. Commitez, mergez `master` avec cette branche, placez votre tag `c0.0`, et revenez sur la branche `code_gen`.



6. *Mergez* avec la branche `ast`, et corrigez les conflits.

7. Corrigez les erreurs de types et complétez la génération de code à l'aide de la commande assembleur `DiviNb` pour la division.

8. Testez, *commitez*, *mergez* `master` avec cette branche, puis placez votre *tag* `c0.1`.



--- * ---

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.