Compilation
TP1 : Lexeur/Parseur
version C (Flex et Bison)

16 août 2022

Prérequis : ce TP suppose vous avez fait de TP0. Si vous pensez être suffisamment à l'aise avec git (notamment avec les branchements), vous pouvez commencer de 0, mais il vous faut créer un dépôt.

Branches : Si vous ne l'avez pas déjà fait, il vous faut créer 4 branches, en plus de `master`, appelées `parser_work`, `parser`, `ast` et `code_gen`.

Exercice 1 (Un parseur sans AST).
Attention, avec flex/bison, la séparation lexeur/parseur semble différente du cours : on fait le maximum dans le parseur, et le lexeur n'est en apparence utilisé que pour les tokens non triviaux ; cela n'empêche pas que l'on passe toujours par le lexeur, c'est juste que les symboles uni-caractères sont associé à un token représenté par eux-même.

1. Placez-vous dans `parser_work`.
2. Modifiez votre `main.c` ainsi [1] en conservant un maximum de lignes : [2]

```c
/* file main.c
 * compilation: gcc -o compiler main.c parser.tab.c lexer.tab.c
 * result: executable
 */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
if (!yyparse()) {                    // call to the parsing (and lexing) function
  printf("\nParsing:: c'est bien une expression arithmétique\n");  // reached if parsing follow
}
exit(EXIT_SUCCESS);
}
```

   Dans ce programme, on se contente d'essayer de parser, et d'afficher si l'analyse syntaxique à réussie ou échouée.
   La fonction `yyparse()` est générée par bison dans le code du parseur (voir questions suivantes). Cette fonction retourne 0 lorsque les analyses sémantiques et syntaxiques terminent avec succès, 1 lorsqu'il y a une erreur lexicale ou syntaxique, et 2 s'il y a une erreur inattendue.
3. Créez un fichier de génération de parser : `parser.y`
   On y définit le parseur en utilisant des `char` comme TOKEN :

```
/* file parseur.y
 * compilation: bison -d parseur.y
```

1. si vous n'en avez pas, créez-en un
2. Lorsque vous êtes versionné, n'effacez pas une ligne pour la remettre, vous risquez d'ajouter un espace et de perdre une partie de l'historique

```
 * result: parseur.tab.c = C code for syntaxic analyser
 * result: parseur.tab.h = def. of lexical units aka lexems
 */

%{  // the code between %{ and %} is copied at the start of the generated .c
 #include <stdio.h>
 int yylex(void);          // -Wall : avoid implicit call
 int yyerror(const char*); //         on fonctions defined by the generator
%}

%token NUMBER      // kinds of non-trivial tokens expected from the lexer
%start expression  // main non-terminal

%% // denotes the begining of the grammar with bison-specific syntax

expression:              // an expression is
   expression '+' term   // either a sum of an expression and a term
 | expression '-' term   // or an expression minus a term
 | term                  // or a term
 ;

 term:                   // a term is
   term '*' factor       // either a product of a term and a factor
 | factor                // or a factor
 ;

factor:                  // a factor is
   '(' expression ')'    // either an expression surounded by parentheses
 | '-' factor            // or the negation of a factor
 | NUMBER                // or a token NUMBER
 ;

%% // denotes the end of the grammar
   // everything after %% is copied at the end of the generated .c
int yyerror(const char *msg){ // called by the parser if the parsing fails
  printf("Parsing:: syntax error\n");
  return 1;                   // to distinguish with the 0 retured by the success
}
```

Dans l'ordre :
— on définit le prototype du lexeur et de la fonction d'erreur de bison [3]
— on y décrit un unique token non trivial (càd qui n'est pas un simple `char`) appelé `NUMBER`,
— on y décrit une grammaire avec `expression`, `term` et `factor` comme non terminaux et avec `'+'`, `'-'`, `'*'`, `'/'`, `'('`, `')'`, `'-'` ainsi que le token `NUMBER` comme terminaux.
— on inclue des bibliothèques pour pouvoir écrire le programme renvoyé en cas d'erreur,
— on y décrit la fonction `yyerror` qui est appelée en cas d'erreur

4. La grammaire utilisée est un peu complexe. C'est parce qu'elle doit être non ambiguë. Heureusement, *bison*, le générateur de parseur, nous permet d'avoir des grammaires ambiguës pourvu que l'on fournisse des règles de priorité et dassociativité pour désambiguïser.
Retournez dans `parseur.y` et modifiez ainsi le fichier :

3. Il s'agit d'éviter certains `warning` à la compilation : cf. `https://stackoverflow.com/questions/20106574/simple-yacc-grammars-give-an`

```
%token NUMBER
%start expression

%left '+' '-'
%left '*'
%nonassoc UMOINS

%%


expression:
    expression '+' expression
  | expression '-' expression
  | expression '*' expression
  | '(' expression ')'
  | '-' expression %prec UMOINS
  | NUMBER
  ;
```

Cette version fait exactement la même chose que la précédente, mais en plus concis et intuitif grâce aux lois de priorité et d'associativité :

— Les règles d'associativité sont indiqués par `%left` ou `%right`.

— Les règles de priorité sont implicites : `'*'` est prioritaire sur `'+'` et `'-'` car la ligne "`%left '+' '-'`" est placée avant la ligne "`%left '*'`".

— Lorsque l'associativité de fait aucun sens (par exemple pour un opérateur unaire) mais que l'on veut avoir une priorité, on utilise `%nonassoc` et on place les opérateur au bon niveau.

— Lorsqu'un token est utilisé dans plusieurs règles avec des priorités/associativités différentes (comme `'-'`), on utilise une balise pour indiquer la priorité d'une des règles, ici la seconde règle du moins est balisée `UMOINS` qui a une autre priorité que `'-'`.

5. Créez un fichier de génération de lexeur : `lexeur.l`
   On y définit l'unique token non trivial :

```
/* file lexeur.l
 * compilation: flex lexeur.l
 * result: lex.yy.c = lexical analyser in C
 */

%{
 #include <stdio.h>        // printf
 #include "parseur.tab.h" // token constants defined in parseur.y via #define
%}

%%

0|[1-9][0-9]*
                { printf("lex: création token NUMBER %s\n",yytext);
                  return NUMBER; }
[ \t\r]
                { ; } // separator
\n
                { printf("lex: fin de lecture");
                  return 0; }
.
```

```
                { printf("lex: création token %s\n",yytext);
                  return yytext[0]; }

%%
int yywrap(void){ return 1; } // function called at the end of the file
```

Dans l'ordre :
- on inclut `parseur.tab.h` qui est généré à partir de `parseur.y` et qui définit le token `NUMBER`,
- `0|[1-9][0-9]*` est l'expression régulière capturant les entiers,
- la partie "`printf("lex: création token NUMBER %s\n",yytext); return NUMBER;`" est l'action associée à l'expression régulière : lorsque le lexeur a reconnu l'expression en question il va donc afficher "`création token NUMBER %s`" sur le terminal où `%s` est remplacé par le lexème reconnu, puis il va envoyer le token `NUMBER` au parseur avant de continuer à chercher le prochain lexème,
- la ligne `[ \t] {;}` permet d'ignorer les séparateurs (ici l'espace, le retour chariot et la tabulation),
- la ligne `\n {return 0;}` permet d'arrêter le lexeur (et donc le parseur) au premier retour à la ligne,
- la ligne `. {return yytext[0];}` dit que si on lit autre chose, on renvoi au parseur un token trivial avec ce caractère,
- la fonction `yywrap` est appelée à la fin du fichier, elle doit toujours renvoyer 1.

6. Compilez tout ça à l'aide des trois commandes suivantes dans le terminal :

```
$ bison -d parseur.y
$ flex lexeur.l
$ gcc -o main main.c parseur.tab.c lex.yy.c
```

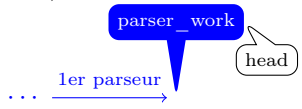
   Cela génère deux fichiers intermédiaires : votre parseur `parseur.tab.c`, votre lexeur `lex.yy.c`, puis votre exécutable `main`

7. Vous pouvez lancer `.\main` dans un terminal. Entrez une expression arithmétique, vous verrez les *log*s du lexeurs, puis, si l'expression entrée est correcte vous aurez un message l'indiquant sinon vous aurez un message d'erreur.

8. Vous pouvez maintenant mettre à jour votre `Makefile` et ajouter ces 4 fichier au suivit de git, *commit*er et *push*er :

```
$ git add lexeur.l parseur.y main.c Makefile
$ git commit -m "1er parseur"
$ git push
```

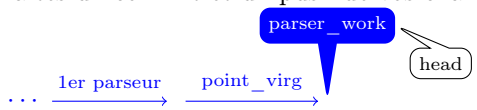
   Remarquez que l'on ne *commit*e pas les fichiers générés. Il ne faut jamais *commit*er les fichiez générés car ils changent beaucoup, et que l'on peut les retrouver à partir des fichiers originaux. Voici l'historique obtenu. Pour plus de lisibilité, on ignore la partie de l'historique venant du TP1, et on affichera les autres branches lorsqu'on les modifiera :



9. On voudrait ne reconnaître que des expressions finissant par un "`;`", car elles seules sont des commandes exécutable en *JS*. Pour ça, on va modifier `parseur.y` :
   - ajouter un non-terminal `commande` avant le non-terminal `expression`,
   - ce non terminal doit reconnaître expression suivie d'un "`;`", ce que l'on écrit :

```
commande :   expression ';'
```

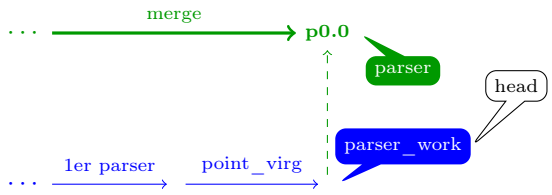
— changez le non-terminal principal `%start expression` pour qu'il attende une commande. Faites un commit et un push de vos changements :



Exercice 2 (Fragment `0.1` du parser).

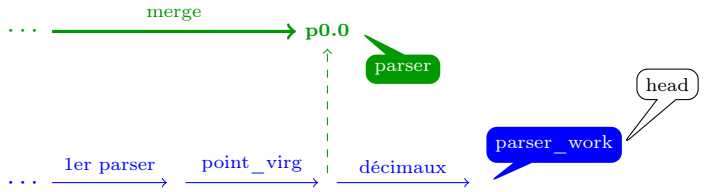
1. Faites un commit, placez-vous sur `parseur`, *merge*z ce que vous avez fait et *tag*gez la version puis revenez sur la branche de travail :

```
$ git switch parser
$ git merge parser_work
$ git tag -a p0.0 -m ``premiere version faites en TP par <mon nom>''
$ git push --tags
$ git checkout parser_work
```

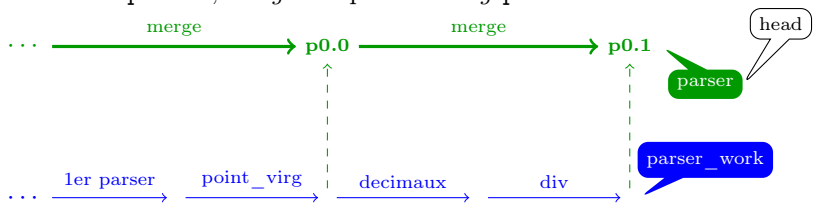


Remarque : pensez à *push*er avec l'option `-tags`, sans ça vos *tags* ne seront pas enregistrés sur le dépôt. [4]

2. Rajoutez une écriture décimal (virgule fixe) pour nos nombres en modifiant l'expression régulière correspondante dans le fichier `lexeur.l`. Attention, il s'agit bien de remplacer les entiers par les flottants : en JS il n'y a pas de *int*, seuls les flottants existent. Faites un nouveau commit. La situation de votre dépôt devrait alors être la suivante :



3. Rajoutez l'opérateur de division `/`. Faites attention à sa priorité ! Faites un commit, revenez sur la branche `parser`, *merge*z 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 d'implémenter variables et fonctions que de faire un compilateur vers notre assembleur abstrait. Néanmoins, pour les tout premiers fragments, il se trouve que c'est assez simples, et on va le faire pour se familiariser avec l'outil de parsing.

4. Je crois qu'il s'agit d'un soucis de rétro-compatibilité : les très vieux dépôts ne gèrent pas les tags, l'option renverra alors une erreur.

1. Créez une nouvelle branche appelée `interpreter`, qui part de `p0.0` :

   ```
   $ git branch interpreter p1.1
   $ git switch interpreter
   ```

2. Commencez par modifier le fichier de génération du lexeur afin qu'il transmette les valeurs des entiers lus (pour l'instant il ne fait que dire qu'il a vu un entier...). Pour ça, on modifie l'action du lexème de `NUMBER` afin de passer l'entier reconnu :

   ```
   { printf("lex: création token NUMBER %s\n",yytext);
     yylval=atoi(yytext);
     return NUMBER; }
   ```

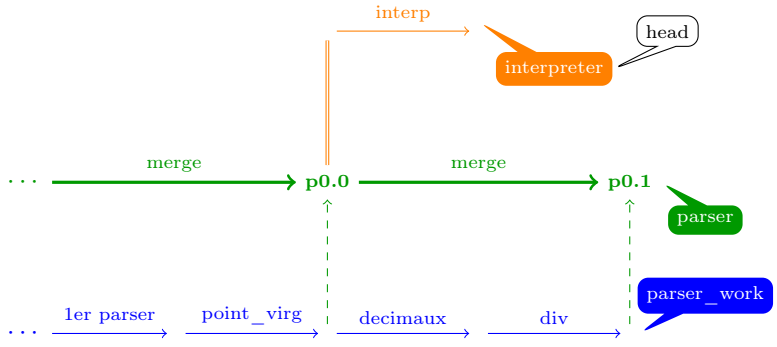
   la variable `yylval` est celle qui récupérera le contenu des token à leur création, la variable `yytext` est celle qui contient le lexème lu (c'est donc une string).

3. Ensuite il faut exprimer les contenus créés à chaque règle :

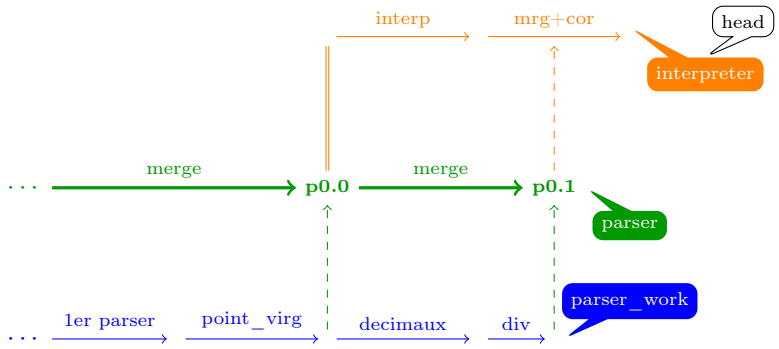
   ```
   commande:
       expression ';'
                    { printf("Resultat= %i\n", $1); }

   expression:
       expression '+' expression
                    { $$ = $1+$3; }
     | expression '-' expression
                    { $$ = $1-$3; }
     | expression '*' expression
                    { $$ = $1*$3; }
     | '(' expression ')'
                    { $$ = $2; }
     | '-' expression %prec UMOINS
                    { $$ = -$2; }
     | NUMBER
                    { $$ = $1; }        // default semantic value
     ;
   ```

   Dans les actions (entre les accolades), `$1,$2,$3` désignent le contenu du premier, second et troisième token utilisé dans la règle. Les actions, ici, sont de type entier, c'est l'entier que l'on va mettre dans le token créé.
   Remarquez le retour à la ligne et 'indentation de l'action ; celle-ci n'est pas obligatoire, mais elle permet deux choses : de permettre à git de plus facilement suivre les changements (par exemple le *dif*, ici ne fera qu'ajouter des choses), et de permettre d'aligner les actions même lorsque la règle est complexe.

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



5. Faites le merge avec `p0.1`.[5] Il y aura peut être un conflit car vous avez ajoutés deux lignes au même endroit (typiquement, l'action du `*` dans `interp` et la règle de `/`). Réglez le commit à la main en arrangeant correctement les deux lignes. (Rappel : après avoir modifié le fichier conflictuel, faites un `git add` et un `git commit`).



6. Essayez de compiler votre version et de la tester avec des flottants. Surprise : on n'a pas le bon résultat.
   Attention, on a *commité* une version incorrecte. Ce n'est pas une branche de *realease*, donc ce n'est pas très grave, c'est même normal après un merge non trivial.
   L'erreur vient d'abord de `lexer.l` : on utilise `atoi` au lieux de `atof`,mais ce ne sera pas suffisant : Par défaut les tokens de bison ont le type `int`, mais ici, on veut utiliser des `double` associés aux tokens il faut, pour ça, ajouter un type union dans `parseur.y` avec :

   ```
   %union { double number } ;
   %token <number> NUMBER
   %type <number> expression
   ```

   Sont décrits ci-dessus les types qui seront utilisés (ici `number` qui désigne les *doubles*) ainsi que le types des contenus du token `NUMBER` et du non-terminal `expression` (on parle du type de leur valeur sémantique). On utilise le mot clé `%union` car ou pourra par la suite avoir des valeurs mixtes en fonction des tokens et des non-teminaux.
   Il faut aussi modifier le lexeur en conséquence : l'action du lexème des nombres devient

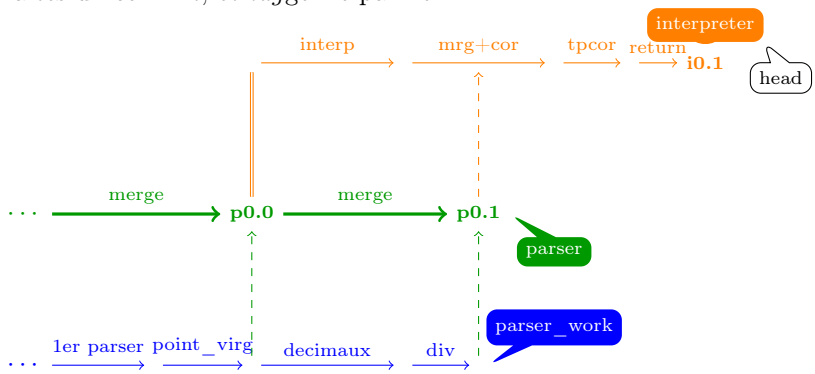
   ```
   { printf("lex::NUMBER %s\n",yytext); yylval.number=atof(yytext); return NUMBER; }
   ```

   On peut maintenant vérifier que tout est bon et *commit*er la correction.

7. Dans cette version, on a un petit peu triché : en effet, si vous regardez bien `main.c` et `parser.y`, vous verrez que l'affichage du résultat ne se fait pas dans le main, comme on pourrait s'y attendre, mais dans la dernière action du parser.
   Pour pouvoir récupérer le résultat dans le *main* et pouvoir l'y afficher, il faut faire quelques acrobaties car le parser de *bison* ne retourne pas de valeur mais il peut prendre un pointeur en entrée :
   — Dans le main, créez une variable `rez` de type `double` dont vous donnerez la référence à la fonction `yyparse` et que vous utiliserez pour afficher le double récupéré.
   — Dans `parseur.y` ajoutez l'option `%parse-param {double *rez}` qui permet de récupérer l'entrée de la fonction `yyparse`, puis modifiez l'action de la seule rêgle de `commande` pour qu'elle mette son résultat dans `rez`. Il faut aussi changer la signature de `yyerror` qui prend maintenant un agument de type `double*` en plus.

5. Remarque : on peut aussi, de manière équivalente, faire le merge avec `parser`.

Faites un commit, et *tag*gez le par `i0.1`.



Dans les exercices suivants, on oubliera la branche "interpreter". En effet, l'interpréteur n'est pas demandé pour le projet contrairement au compilateur. En fait, l'interpréteur devient vite difficile à écrire une fois que l'on parle de variable, très difficile lorsque l'on introduit les fonctions/clôtures, et encore plus avec les exceptions.

Exercice 4 (Créer un AST en sortie).
Avant de pouvoir générer notre code assembleur, on veut pouvoir manipuler notre arbre syntaxique, et pour ça on utilisera une simplification de ce dernier : l'AST. Ce n'est pas strictement nécessaire, surtout au début, mais c'est très pratique pour s'y retrouver et séparer les problèmes.

1. Allez sur la branche `ast` et faites un merge avec le *tag* `p0.0`.
2. Téléchargez les fichiers `AST.h` et `AST.c`.
   Vous y trouverez la structure de l'AST dans `AST.h`, et les fonctions pour la manipuler implémentées dans `AST.c`.[6] ajoutez le au suivi git et faites un *commit*.
3. Dans `lexeur.l`, modifiez l'action du lexème des nombres :

   ```
   { printf("lex::NUMBER %s\n",yytext); yylval.number=atoi(yytext);
     return NUMBER; }
   ```

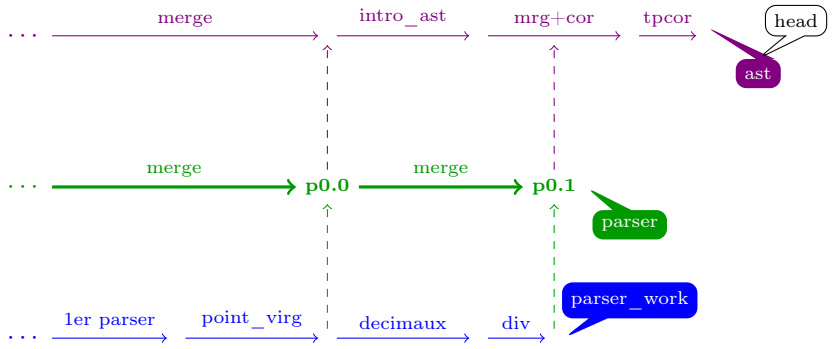
   Remarquez qu'il s'agit d'une version intermédiaire aux deux versions de l'exercice précédent : on a toujours des entiers (puisque l'on utilise `atoi`) mais on doit tout de même préciser `.number` car on va utiliser un `%union` dans le parser afin de manipuler nos *AST*s.
4. Nous allons utiliser le type `AST_comm` comme type de retour des expressions puisque l'on s'attend à lire une commande, ce qui demande les mêmes manipulations qu'à l'exercice précédent :
   — inclure `AST.h` dans `parser.y` et `main.c`,
   — ajouter l'option `%parse-param {AST_comm *rez}` dans `parser.y`,
   — changez la signature de `yyerror`,
   — déclarez un `AST_comm` dans `main.c` dont vous passerez la référence à `yyparse` puis que vous affichez grâce à `print_comm`.
5. Comme à l'exercice précédent, on rajoute un type union dans `parser.y`, mais un peu plus complexe cette fois, car le token `NUMBER` transporte toujours un entier alors que le non-terminal `expression` transporte, lui, un *AST* :
6. il faut maintenant écrire les actions associées à la grammaire permettant de construire un AST. :

   ```
   commande:
       expression ';'
                       { *rez = new_command($1); }
   ```

6. C'est une proposition, vous n'êtes pas obligés d'utiliser cette version.

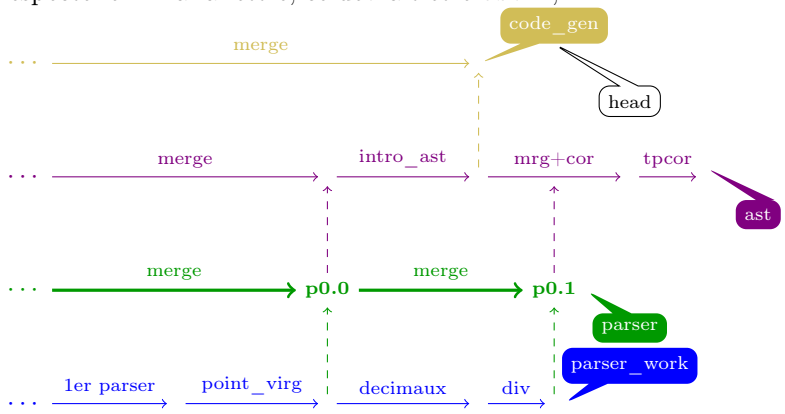
```
expression:
    expression '+' expression
                    { $$ = new_binary_expr('+',$1,$3); }
  | expression '-' expression
                    { $$ = new_binary_expr('-',$1,$3); }
  | expression '*' expression
                    { $$ = new_binary_expr('*',$1,$3); }
  | '(' expression ')'
                    { $$ = $2; }
  | '-' expression %prec UMOINS
                    { $$ = new_unary_expr('M',$2); }
  | NUMBER
                    { $$ = new_number_expr($1); }
  ;
```

7. Vous pouvez tester et faire un commit (appelé `intro_ast` dans la suite) si tout va bien.
8. *Mergez* avec `p0.1`. Comme dans l'exercice précédent, vous aurez peut-être un petit conflit à gérer, puis des erreurs de gestion des flotants :



Exercice 5 (Génération de code).

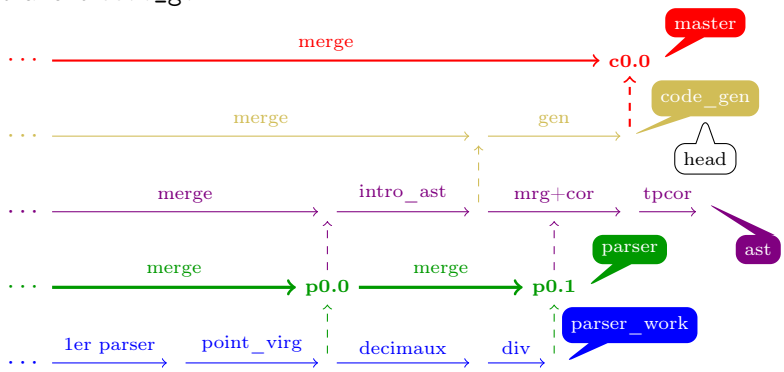
1. Revenez sur `code_gen` et *mergez*-le avec la version de `ast` issue de `intro_ast`, si vous avez respecté le TP à la lettre, ce devrait être `ast^^`,



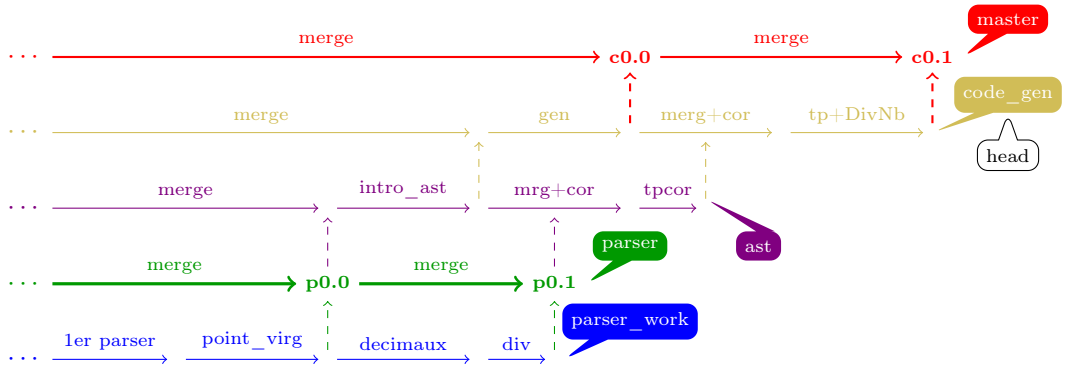
2. Dans `AST.c`, créez une nouvelle fonction d'affichage `code` qui va parcourir l'arbre et va faire un affichage post-fixe de sa structure en affichant : [7]
   — la ligne "`CsteNb` *n*" après avoir quitté un noeud `('N',n,NULL,NULL)`,

7. Passez à la ligne après chaque affichage pour plus de lisibilité.

— la ligne “`AddiNb`” après avoir quitté un noeud `('+',0,t1,t2)`,
— la ligne “`MultNb`” après avoir quitté un noeud `('*',0,t1,t2)`,
— la ligne “`SubiNb`” après avoir quitté un noeud `('-',0,t1,t2)`,
— la ligne “`NegaNb`” après avoir quitté un noeud `('M',0,t1,NULL)`,
Testez votre code généré à l'aide de la miniJSMachine.
*Commit*ez, allez sur `master` et *merge*z cette branche, placez votre tag `c0.0`, et revenez sur la branche `code_gen` :



3. *Merge*z avec la branche `ast`, et corrigez les conflits.
4. Corrigez les erreurs de types et complétez la génération de code à l'aide de la commande assembleur `ModuNb` pour le modulo.
5. Testez, *commit*ez, *merge*z `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.
Seule exception : l'exercice ci-dessous qui explique comment changer vos entrée-sorties pour prendre et renvoyer des fichier. Vous pouvez le faire quand vous voulez, mais ça deviendra vite nécessaires affin de tester sur des codes de plusieurs lignes.

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 déboguer, mais aussi pour travailler en parallèle avec des vitesses différentes. N'hésitez pas non plus à poser d'autres tags afin de pouvoir *merge*r un point particulier tout en continuant sur une branche.

Exercice 6 (Prendre un fichier en entrée).

— dans `lexeur.l` : supprimez la ligne sur `\n` et ajoutez le retour à la ligne parmi les séparateurs, c'est maintenant la fin de fichier qui quitte le lexeur,
— dans `main.c` : placez un pointeur de fichier vers le chemin d'accès (donné en premier argument de votre programme) dans la constante global `yyin` utilisée par le lexeur : [8]

```
extern FILE* yyin;
yyin = fopen(args[1],"r");
```

8. par défaut `yyin` contient l'entrée standard, c'est pour ça que l'on n'avait pas besoin de le faire avant