Compilation
TP1 : Lexeur/Parseur
version OCaml (Ocamllex et OCamlyacc)

Flavien Breuvart

janvier 2025

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

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

```
(*fichier main.ml *)
let _ =                                              (*main en OCaml*)
  try
    let lexbuf = Lexing.from_channel stdin in        (*lexeur lancé sur stdin*)
    while true do                                    (*on ne s'arrête pas*)
      Parseur.main Lexeur.token lexbuf               (*parseur une ligne*)
    done
  with
   | Lexeur.Eof           ->  exit 0                 (*impossible*)
   | Lexeur.TokenInconnu                             (*erreur de lexing*)
   | Parsing.Parse_error ->                          (*erreur de parsing*)
      Printf.printf ("Ceci n'est pas une expression arithmetique\n")
```

plus précisément:

- le `while true do` permet d'essayer de parser chaque lignes entrées en boucle jusqu'à avoir une erreur,
- la ligne "`let lexbuf = Lexing.from_channel stdin in`" crée un scanner ,[3]
- la ligne "`Parseur.main Lexeur.token lexbuf`" appel le parseur avec le lexeur et le scanner comme arguments,[4]

[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

[3] que l'on appelle traditionnellement lexbuf, car il s'agit du fichier à lexeur mis dans un buffeur.

[4] Pour ceux qui ont suivi le cours de M.Leroux l'an dernier : normalement on devrait donner l'entrée au lexeur qui calcule la liste de token qui est donnée au parseur, mais ici c'est le parseur qui va paresseusement lancer le lexeur pour récupérer les tokens un à un.

- la première exception `Lexeur.Eof` n'est pas utile ici: il n'y aura pas de symbole de fin de fichier en lisant le stdin, mais je la met pour ne pas que vous l'oubliez plus tard...
- les deux autres exceptions reportent les blocages du lexeur et du parseur. Notez qu'il s'agit d'un catch multiple: les deux exceptions vont afficher le même printf. N'hésitez pas à faire des affichages séparés dans votre version.[5]

3. Créez un fichier de génération de parser : `parseur.mly`

```
%token NUMBER PLUS MINUS TIMES GPAREN DPAREN EOL
%type <unit> main expression terme facteur
%start main
%%
main:
   expression EOL                {}
 ;
expression:
   expression PLUS terme         {}
 | expression MINUS terme        {}
 | terme                         {}
 ;
terme:
   terme TIMES facteur            {}
 | facteur                       {}
 ;
facteur:
   GPAREN expression DPAREN      {}
 | MINUS facteur                 {}
 | NUMBER                        {}
 ;
```

   Dans l'ordre:

   - on y déclare nos noms de tokens,
   - on déclare le type (ici `unit`) de nos non-terminaux,
   - on déclare le non-terminal principal (ici `main`),
   - on y décrit une grammaire, si elle est si compliquée, c'est pour forcer les priorités.
   - les accolades `{}` indiquent qu'il n'y a aucune action résultante de la lecture de ces patterns.

4. La grammaire utilisée est un peu complexe. C'est parce qu'elle doit être non ambiguë. Heureusement, *ocamlyacc*, 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.mly` et modifiez ainsi le fichier :

```
%token NUMBER PLUS MINUS TIMES GPAREN DPAREN EOL

%left PLUS MINUS
%left TIMES
%nonassoc UMINUS

%type <unit> main expression
```

[5] Le report d'erreur de compilation n'étant pas considéré dans ce projet, vous pouvez renvoyer ce que vous souhaitez.

```
%start main
%%
main:
   expression EOL
         {}
 ;
expression:
   expression PLUS expression
         {}
 | expression MINUS expression
         {}
 | expression TIMES expression
         {}
 | GPAREN expression DPAREN
         {}
 | MINUS expression %prec UMINUS
         {}
 | 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: `TIMES` est prioritaire sur `PLUS` et `MINUS` car `%left PLUS MINUS` est définit avant `%left TIMES`.
- Lorsque l'associativité de fait aucun sens (par exemple pour un opérateur unaire) mais qui 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 `MINUS`), on utilise une balise pour indiquer la priorité d'une des règles, ici la seconde règle du moins est balisée `UMINUS` qui a une autre priorité que `MINUS`.

5. Créez un fichier de génération de lexeur : `lexeur.mll`, on définit les tokens:

```
(*fichier lexeur.mll *)
{
  open Parseur
  exception Eof
  exception TokenInconu
}
rule token = parse
  [' ' '\t' '\r']
      { token lexbuf }
| ['\n']
      { EOL }
| ['0'-'9']+
      { NUMBER }
| '+'
      { PLUS }
| '-'
      { MINUS }
```

```
| '*'
      { TIMES }
| '('
      { GPAREN }
| ')'
      { DPAREN }
| eof
      { raise Eof }
| _
      { raise TokenInconu }
```

Dans l'ordre:

- on inclue `Parseur` qui est sera généré à partir de `parseur.mly` et qui définit les tokens (avec leur types),
- la ligne `[' ' '\t'] {token lexbuf}` reconnaît les espaces et tabulations mais ne crée pas de token car ce sont des *séparateurs*; le "token lexbuf" est simplement une façon de dire au lexeur de continuer.
- la ligne `['\n'] {EOL}` crée le token `EOL` en cas de changement de ligne,
- le `['0'-'9']+` est l'expression régulière capturant les entiers,
- juste après le "`{NUMBER}`" est l'action associée, ici on retourne simplement le token `NUMBER`,
- les 6 lignes suivantes récupère les symboles d'opération et y associe les tokens correspondants,
- les deux dernières lignes lancent des exceptions (récupérées dans `main.ml`) si on lit la fin du fichier ou un symbole inconnu.

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

```
$ ocamllex lexeur.mll
$ ocamlyacc parseur.mly
$ ocamlc -c parseur.mli lexeur.ml parseur.ml main.ml
$ ocamlc -o main lexeur.cmo parseur.cmo main.cmo
```

   Cela génère plusieurs de fichiers intermédiaires, parmi lesquels votre lexeur `lexeur.ml`, votre parseur `parseur.ml`, et votre éxécutable `main`. Attention à l'utilisation de `ocamlc` :l'ordre des arguments est important car la compilation de chaque ne peut utiliser que les dépendances déclarées précédament.

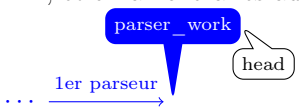
7. Vous pouvez lancer `main` dans un terminal et taper une expression avant d'aller à la ligne. Si elle est correcte rien ne se passe mais si elle est fausse vous aurez un message d'erreur.

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

```
$ git add lexeur.mll parseur.mly main.ml 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 :



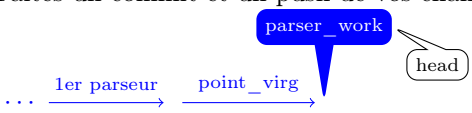
9. On voudrait ne reconnaître que des expressions finissant par un "`;`", car elles seules sont des commandes exécutable en *JS*. Pour ça, on va modifier `parseur.mly` :

   - ajoutez un token reconnaissant ";" (dans `lexer.mll` mais aussi dans `parser.mly`),
   - ajoutez un non-terminal `commande` entre les non-terminaux `result` et `expression`,
   - ce non terminal doit reconnaître expression suivie d'un ";", ce que l'on écrit :

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

   - changez la definition de `result` pour qu'il pour qu'il attende une commande au lieu d'une expression.

   Faites un commit et un push de vos changements :

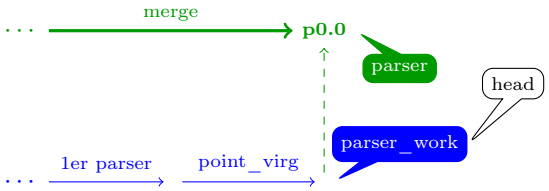


--- * ---

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 :

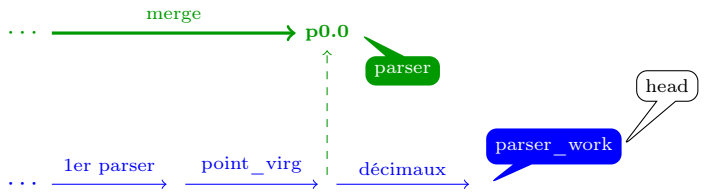
   ```
   $ git switch parser
   $ git merge parser_work
   $ git tag -a p0.0 -m ``premiere version faites en TP par <mon nom>''
   $ git push --tags
   $ git checkout parser_work
   ```



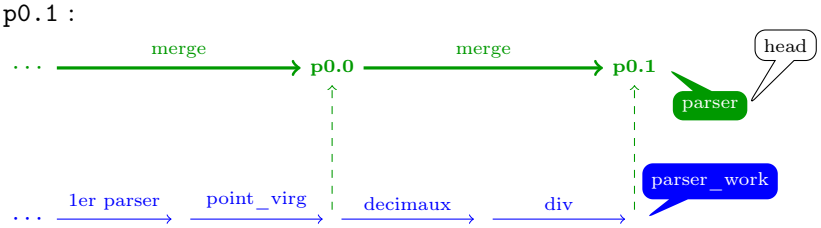
   Remarque : pensez à *push*er avec l'option `-tags`, sans ça vos *tags* ne seront pas enregistrés sur le dépôt.

2. Rajoutez une écriture décimal (virgule fixe) pour nos nombres en modifiant l'expression régulière correspondante dans le fichier `lexeur.mll`. 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 `/`. Essayez d'avoir la bonne priorité (même si vous ne pouvez pas encore le tester). Faites un commit, revenez sur la branche `parser`, *mergez* 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. 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 :

```
| ['0'-'9']+
    as lexem
    { NUMBER(int_of_string lexem) }
```

La ligne `as lexem` permet de créer une variable `lemem` innitialisée avec le lèxem reconnu. Celui-ci sera alors donné en argument au token `NUMBER` (après avoir été traduit en entier).

3. Puis on modifie la première ligne du parseur afin de lui signaler que les tokens `NUMBER` ont maintenant un contenu :

```
%token <int> NUMBER
%token PLUS MINUS TIMES GPAREN DPAREN EOL
```

4. Après ça, il faut dire au parseur que les non-teminaux aussi vont créer des tokens contenant des entiers :

```
%type <int> main expression
```

(attention cette ligne remplace une ligne existante, je suis sure que vous allez trouver laquelle).

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

```
main:
   commande EOL
         { $1 }
 ;
commande:
   expression SEMICOL
         { $1 }
 ;
expression:
   expression PLUS expression
         { $1+$3 }
 | expression MINUS expression
         { $1-$3 }
 | expression TIMES expression
         { $1*$3 }
 | GPAREN expression DPAREN
         { $2 }
 | MINUS expression %prec UMINUS
         { -$2 }
 | NUMBER
         { $1 }
 ;
```

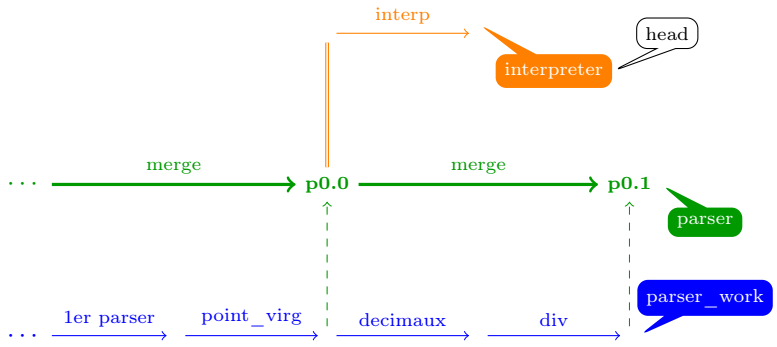
   Dans les actions (entre les acolades), `$1,$2,$3` désignent le contenu du premier, second et troisième token utilisé dans la règle. Les actions, ici, sont de type entier, c'est l'entier que l'on va mettre dans le token créé.

6. Enfin dans `main.ml`, on va afficher l'entier trouvé :

```
Parseur.main Lexeur.token lexbuf            (*parseur une ligne*)
|> Printf.printf "%i\n%!";
```

   La fonction `Parseur.main`, avec le lexeur et l'entrée standard en arguments, retourne l'entier calculé, on affiche celui-ci avant de passer à la ligne et de flush (indiqué par `%!`, force l'affichage des caractères mis en buffer).

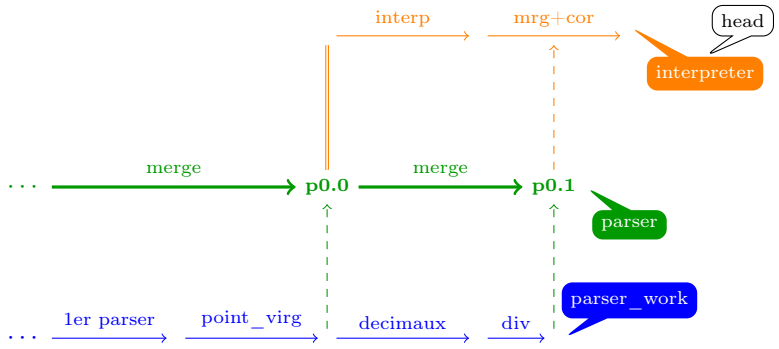
7. Vous pouvez tester et faire un commit (appelé `interp` dans la suite) si tout va bien.



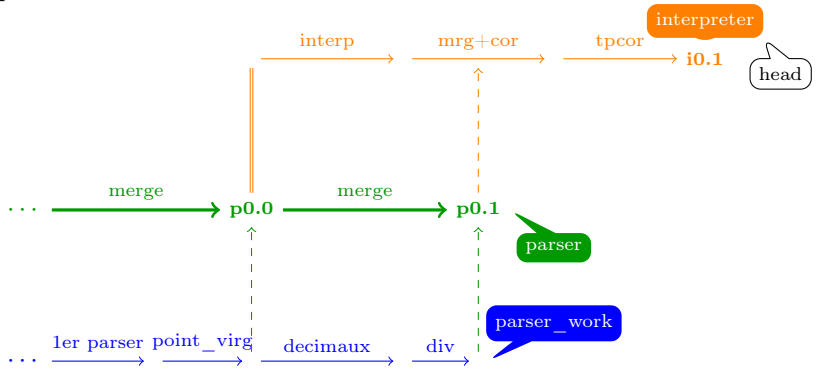
8. Faites le merge avec `p0.1`.[6] Il y aura peut être un conflit car vous avez ajoutés deux lignes au même endroit (typiquement, l'action du `TIMES` dans `interp` et la règle de `DIV`). Réglez le commit

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

à la main en arrangeant correctement les deux lignes. (Rappel : après avoir modifié le fichier conflictuel, faites un `git add` et un `git commit`).



9. Essayez de compiler votre version et testez-la sur des flotant. Surprise : ça ne fonctionne pas. C'est normal : même si on peut lire des flotants, on essait de les mettres dans un token (`NUMER`) contenant des entiers.
   Il faut donc changer tous les types entiers de partout pour y mettre des doubles. Attention : en *OCaml*, les opérations sur les entiers et sur les flottants sont différentes : `*` devient `*.`, `+` devient `+.`, `-` devient `-.` et `\` devient `\.`.
   On peut maintenant vérifier que tout est bon et *commit*er la correction; *tag*gez aussi la version 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 faire un interpreteur, on veut pouvoir manipuler notre 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 le module d'AST disponible ici.[7]
   Vous y trouverez la structure de l'AST et une fonction d'affichage (elle est étrange car j'utilise

[7] Si vous êtes avancé en OCaml, vous remarquerez que l'on n'a pas fait un vrais module, c'est juste pour rendre le code plus accessible, n'hésitez pas à faire un commit (d'abord dans une branche de travail !) qui en fait un module propre.

une bibliothèque de formatage qui fait de l'indentation automatique).[8]
Ajoutez cefichier au suivi git et faites un *commit*.

3. Comme dans l'exercice précédent, il vous faut modifier l'action associée à `NUMBER` dans `lexeur.mll` pour conserver l'entier lu dans le token.

4. Comme précédemment, on modifie la première ligne du parseur afin de lui signaler que les tokens `NUMBER` ont maintenant un contenu entier.

5. Après ça, il faut dire au parseur que les non-teminaux vont créer des tokens contenant un AST du bon type :

```
%type <AST.commande_a> main commande
%type <AST.expression_a> expression
```

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

```
main:
   commande EOL
         { $1 }
 ;
commande:
   expression SEMICOL
         { Expr($1)}
 ;
expression:
   expression PLUS expression
         { Plus ($1,$3) }
 | expression MINUS expression
         { Moins($1,$3) }
 | expression TIMES expression
         { Mult ($1,$3) }
 | GPAREN expression DPAREN
         { $2 }
 | MINUS expression %prec UMINUS
         { Neg $2 }
 | NUMBER
         { Num($1) }
 ;
```

   Cette fois on appel les générateurs des type d'`AST` pour créer l'arbre à la volée.

7. Avant de fermer `parseur.mly`, il faut permettre à OCaml de trouver le module d'AST. Pour ça on ajoute les lignes suivantes tout au début du fichier :

```
%{
  open AST
%}
```

   tout ce qui est entre acolades `%{...%}` sera copié au début du parseur généré `parseur.ml`.

8. Enfin dans `main.ml`, on va afficher l'arbre trouvé :

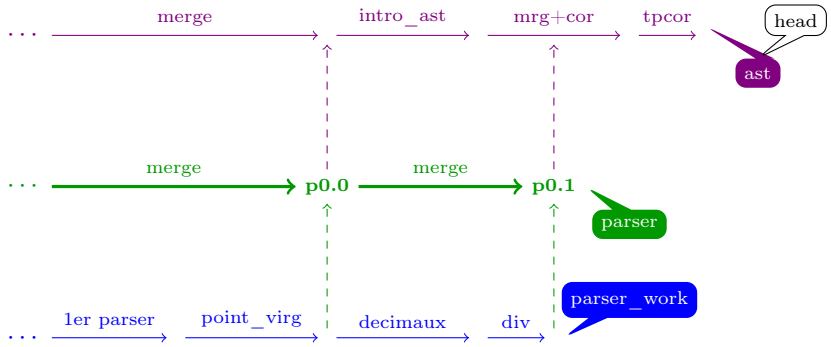
```
Parseur.main Lexeur.token lexbuf           (*parser une ligne*)
|> Format.printf "%a\n%!" AST.print_commande ;
```

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

Le `%a` permet l'afichage d'un élément de type quelconque pourvu que l'on lui donne une fonction d'affichage inteligente avec (ici `AST.print_commande`).

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

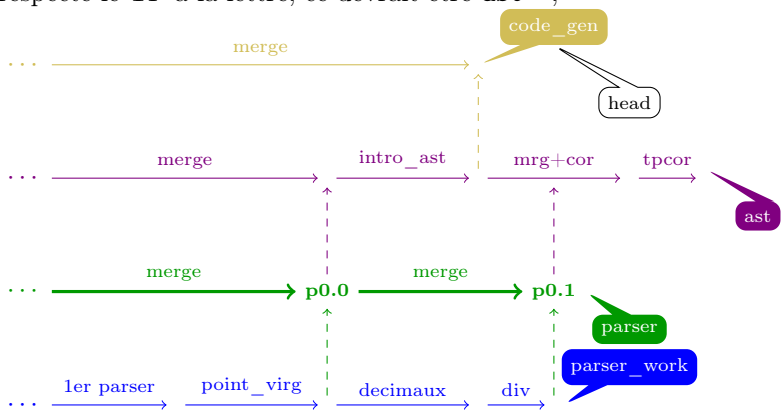
10. *Mergez* avec `p0.1`. Comme dans l'exercice précédent, vous aurez peut-être un petit conflit à gérer, puis des erreurs de gestion des flotants :



--- * ---

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^^`,

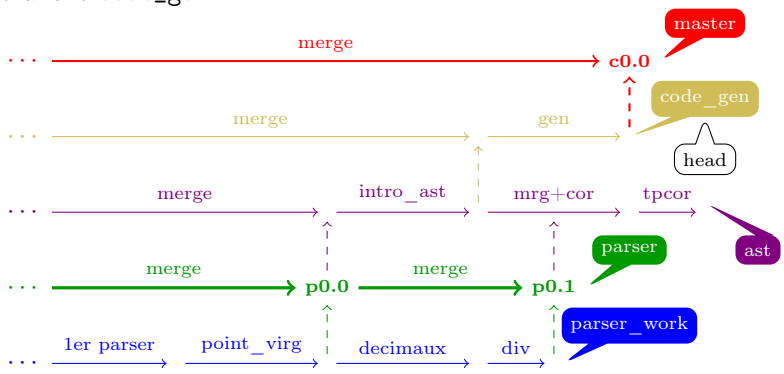


2. Dans `AST.ml`, créez une nouvelle fonction d'affichage `code` qui va parcourir l'arbre et va faire un affichage post-fixe de sa structure en affichant:[9]

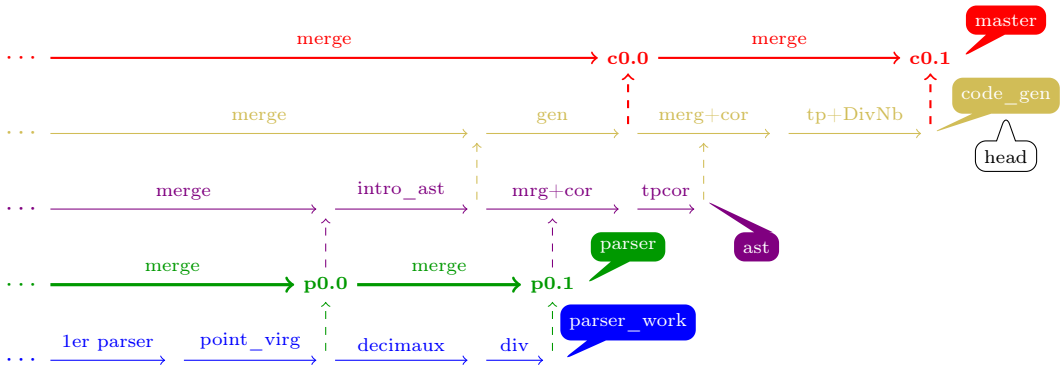
   - la ligne "`CsteNb` n" après avoir quitté un noeud `('N',n,NULL,NULL)`,
   - la ligne "`AddiNb`" après avoir quitté un noeud `('+',0,t1,t2)`,
   - la ligne "`MultNb`" après avoir quitté un noeud `('*',0,t1,t2)`,
   - la ligne "`SubiNb`" après avoir quitté un noeud `('-',0,t1,t2)`,
   - la ligne "`NegaNb`" après avoir quitté un noeud `('M',0,t1,NULL)`,

[9] Passez à la ligne après chaque affichage pour plus de lisibilité.

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 `DiviNb` pour la division.
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 : Manipulation de fichiers

Prendre un fichier en entrée : Ce n'est pas explicitement demandé, mais vous aurrez besoin de pouvoir prendre des fichiers en entrée pour pouvoir tester votre compilateurs sur des exemples raisonables :

- dans `lexeur.mll` : 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.ml` : changez l'entrée standard `stdin` par un canal de lecture sur le fichier dont le noma a été donné en argument `(open_in Sys.argv.(1))`

Nommez un fichier de sortie : Ce n'est pas demandé, mais ça peut être utile :
Au lieu de `printf`, utilisez `fprintf` disponible dans le module `Format` mais aussi dans le module `Printf` avec un type légèrement diférent :

- Si vous utilisez le module `Format`, donnez-lui comme premier argument le formateur `(Format.formatter_of_out_cha (open_out Sys.argv.(2)) )`
- Si vous utilisez le module `Printf`, donnez-lui comme premier argument le canal de sortie `(open_out Sys.argv.(2))`

--- * ---