TP `git` (préalable au cours de compilation)

Flavien Breuvart

15 janvier 2025

Ce TP prend la suite du mini-cours de `git` que vous avez eu en L2 avec Thierry Monteil. On y traite les notions de branche, de merge, de tag et de "workflow" plus en détail. Vous aurez besoin de ces notion pour travailler sur les TPs et le projet du cours de compilation. D'où l'importance de ce TP. Mais ce n'est pas le sujet du cours de compilation, c'est pourquoi on parle de "TP0".

Il est primordial que chacun d'entre vous soit à l'aise avec `git`. En effet, si vous laissez un membre de votre binôme de projet être le "responsable `git`" et faire tous les *commit*s, alors il sera considéré comme l'auteur de l'ensemble du travail, d'autant plus si vous êtes en trinôme. Nous vous conseillons donc de le faire en double, en utilisant chacun votre dépot indépendant, de même pour le TP1, puis de continuer sur un des deux dépôts pour le TP2.

Les exercices 1 et 2 sont à faire AVANT le premier TP. Les autres sont à finir avant le troisième TP (vous avez donc 2 TPs et deux semaines). Quand vous l'avez fini vous pourez passer à une des versions (au choix) du TP1.

Exercice 1 (Mise en place d'un dépot *git*).
Avant tout, corrigeons une conception commune : `git`≠*github*, le premier est un un protocole utilisé par un logiciel libre du même nom permettant le versionnement d'un projet de manière distribuée, l'autre est un serveur de dépôt gratuit (mais de droit privé et racheté par Microsoft) implémentant ce protocole. On utilisera le premier, mais pas le second.

Cet exercice vise à héberger un dépôt `git` sur le *gitlab* de l'université. Vous pouvez le sauter si vous préférez utiliser un dépôt auto-hébergé ou sur un autre héberger. Ce TP suose que vous êtes sous linux, avec `git` installé, et que vous n'utilisez pas d'IDE spécifique (si vous utilisez un ide comme *visual-studio* ou *InteliJ*, vous pouvez l'utiliser pour remplacer les lignes de commandes), mais tout peut se faire aussi sur windows, macOS... en utilisant les outils apropriés.

1. Allez sur https://git.ig-edu.univ-paris13.fr et identifiez vous avec vos identifiant universitaires.
2. Avant de créer le projet, entrez une clé *ssh*, qui permettra d'interagir avec votre dépôt sans entrer votre code à chaque fois. Si vous ne savez pas générer une clé *ssh* sur votre machine, suivez le tuto associé. Allez, dans le menu en haut à droite, dans `Edit profile`, puis dans la colonne de gauche, dans `SSH keys`, entrez la clé *ssh* généré sur votre machine. Il faudra refaire ça pour chaque machine+compte que vous utiliserez (en considérant que tous les ordinateurs des salles de TPs ne sont qu'une seule machine).
3. Revenez à l'accueil du *gitlab* (logo en haut à gauche). et cliquez sur **New project** puis sur **Create a blank project**. Nommez votre projet comme vous le souhaitez (attention, ce dépôt servira potentiellement à héberger votre projet de compilation). L'URL se complète automatiquement, mais vous pouvez en choisir une autre. Ajoutez une description (par exemple "projet de l'UE compilation en L3". Pour la durée du semestre, laissez privé affin d'éviter les fuites... [1]

1. À la fin du semestre, une fois le projet soutenu, vous pouvez le rendre publique ce qui peut être un plus pour votre CV, mais ne sera disponible que pendant la durée de vos études ici. Vous pouvez quasi le faire héberger ailleurs, mais attention, il peut être difficile d'en retirer le contenu et je vous garantie que vous serrez moins fière de vos sculptures de pâte-à-sel dans 10 ans...

Il se peut que vous ayez un onglet “group” à remplir si vous avez été mis dans un groupe par l'enseignant d'un autre cours, dans ce cas ne sélectionnez aucun groupe.

4. Allez sur la page du projet nouvellement créé. Cliquez sur **Clone** et faites une copie du premier lien. Utilisez votre invité de commande en vous plaçant dans le dossier où vous souhaitez enraciner votre projet et tapez :

   ```
   $ git clone lien_que_vous_venez_de_copier
   ```

5. Il vous reste à inviter votre binôme : Sur la page de votre projet du gitlab, dans le menu de gauche, sélectionnez **Project information** → **Members**, puis **Invite Members** en haut à droite. Invitez votre binôme [2] de projet comme `Développer` ou `Maintainer` et chacun de vos enseignants comme `Reporter`.

La suite de ce TP suppose que vous avez initialisé un dépôt *git* avec un commit initial (avec uniquement le *readme*).

Exercice 2 (Initiation à git).
Si vous n'avez jamais utilisé Git, ou que vous êtes rouillés, faites un petit tutoriel ou cours sur les bases. On aura besoin des commandes `git add`, `git commit`, `git push`, `git status` et `tig`.

Vous pouvez regarder les *slides* de T. Monteil (niveau L2) et commencer par le TP associé sections 2 à 10. Ceux-ci vous permettront de vous familiariser avec les manipulation purement locales et monobranches de `git`.

Une autre ressource possible est le livre git-scm, sections 1.3, 1.4, 1.5, 1.7, 2.2 et 2.3 (les sections 1.6 et 2.4 peuvent aussi être utiles). À noter que l'exercice précédent correspond à la section 2.1 et le reste du TP correspond aux sections 2.6, 3.1, 3.2, 3.3 et 3.4 ; néamoins, notre présentation est moins complète et plus focalisés sur les besoin du projet, puisque l'on en profite pour mettre en place celui-ci ; ainsi les deux sont complémentaires.

Exercice 3 (S'y retrouver dans les diférentes branches).
Dans ce TP on va naviguer en trois dimentions : entre les fichiers (pensez "espace"), entre les commits (pensez "temps") et entre les branches (pensez "timelines"). On suppose que vous avez déjà fait l'expérience des deux premières et on insistera sur la troisième.

Si vous êtes perdu entre les commits et les les branchements, utilisez une interface graphique (`gitk`, `git gui` ou `tig`), ou encore un simulateur.

Pour avoir en permanence l'information de la branche sur laquelle vous vous trouvez, nous vous conseillons de modifier l'affichage de votre *bash*. Pour ça, créez, dans votre *home*, un fichier *.bashaliases* [3] et ajoutez-y la ligne :

```
PS1='\[\e[35m\]$(__git_ps1) \[\e[0m\]\$'
```

Vous pouvez maintenant ouvrir un nouveau terminal (ou le rafraîchir avec `$. ~/.bashrc`). Si vous vous placez dans le dossier suivi par le `git` (celui où vous avez votre `.git`), votre prompteur devrait maintenant terminer par `(master) $`, cela vous indique que vous êtes dans la branche "master". [4]

Pour le TP, on n'a pas besoin du chemin d'accès, mais je vous conseil, une fois le TP fini, de le remettre, en plus de la branche :

```
PS1='\[\e[32m\]Dl:\[\e[36m\]$PWD\[\e[35m\]$(__git_ps1)\$ \[\e[0m\]'
```

2. Si vous ne le voyez pas, cela signifie qu'il faut d'abord qu'il se connecte une fois sur le gitlab, cela initialisera son compte.
3. C'est comme un *.bashrc* mais sans modifier ce dernier s'il existe déjà.
4. Selon l'installation de votre dépôt, il se peut que “master” s'appelle “main”, ce n'est pas grave du tout, mais il faudra modifier les commandes suivante en fonction.

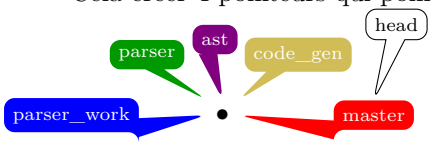
Attention : Une erreur courante sur ce TP consiste à remplacer brutalement un fichier par la correction ou par la version présente sur une autre branche git. Il ne faut jamais faire ça, car l'algorithme de différenciation de `git` n'arrivera pas à suivre et les future *merge* risquent de tous échouer. Pour l'exercice, il faut modifier, à la main, chaque ligne qui a besoin d'être modifiée. Si vous vous êtes trompé sur un branchement, rien de critique, cherchez en ligne comment revenir en arrière avec `git`.[5] Et, dans le pire des cas, vous pouvez toujours refaire un dépôt vu qu'il n'y a rien d'important encore dedans.

Exercice 4 (Mise en place du "*workflow*").
Au début du TP, vous devez vous placer dans le dossier cloné, vous serez automatiquement sur la branche `master`,[6] qui sera la branche de rendu pour la seconde partie compilateur (si vous préférez qu'elle ai un autre nom, vous pouvez le changer).

Créez 4 nouvelles branches que l'on nommera `parser_work`, `parser`, `ast` et `code_gen` :

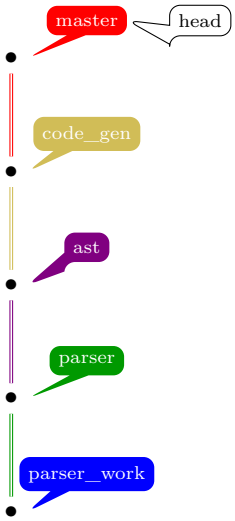
```
(master) $ git branch parser_work
(master) $ git branch parser
(master) $ git branch ast
(master) $ git branch code_gen
```

Cela créer 4 pointeurs qui pointent la même version :



Remarquez la bulle `head`, celle-ci indique la version/branche actuellement suivie. Si vous faites un `git status` vous verrez que vous travaillez actuellement sur la branche `master`.

Notre dessin n'est pas très lisible car on ne voit pas vraiment les "branches" et leurs différences, on va donc informellement ajouter des relations entres elles :



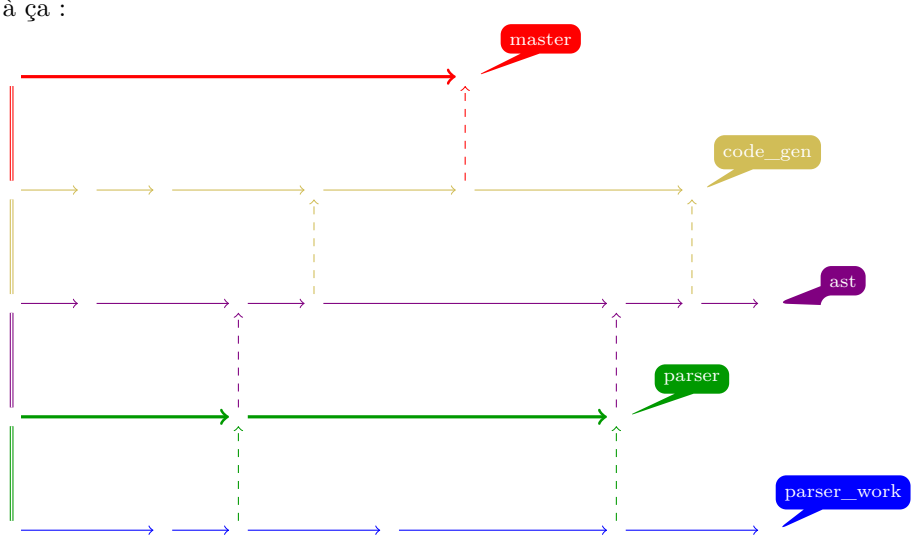
Les doubles lignes indiquent qu'il s'agit de la même version de part et d'autre, mais leur couleur et le fait qu'elles soient marqués au dessus nous permettent de dire qu'elles sont "moralement plus avancées".

En effet, tous les changements faits sur `parser_work` seront à un moment *mergés* dans `parser`, ceux de `parser` seront *mergés* dans `ast`, *etc...* Ainsi, notre flux de travail (*workflow*) devra ressembler

5. l'avantage d'un gestionnaire de version est que rien n'est jamais perdu, par contre l'outil est complexe et il faut apprendre à l'utiliser.

6. Selon le dépôt `git` utilisé, votre branche `master` s'appellera `main`, dans ce cas remplacez simplement "master" par "main" dans toutes les commandes données.

à ça :



— Les lignes pleines horizontales représentent des *commits*, c'est à dire des changements apportés à la version de la branche (la couleur indique la branche).
— Les lignes pointillés verticales indique que l'on a *mergé* la branche d'en dessous avec celle d'au dessus, intégrant ainsi tous les changements de la première dans la seconde (voyez ça comme l'union des changements).
— Les bulles indique les dernière versions versions enregistrés sur les branches en questions, si vous faites un `switch parser` alors vous irez sur la version pointé par `parser`.
— Les branches `parser` et `master` sont plus épaisses, c'est pour indiquer que ce sont des branches de *release*, on ne peut y enregistrer que des versions qui compilent, et c'est sur celles-ci que l'on trouvera les *tags* de rendu.

Attention : Ces conventions ne sont pas standard du tout et chaque présentation de *workflow* `git` utilise des différentes (généralement une présentation qui rend son *workflow* compréhensible).

Ici, on voit que le projet avance sur deux dimensions distinctes :
— vers la droite, avec de nouveaux commits qui représenteront des "fragments" de plus en plus gros du langage de votre compilateur,
— vers le haut, de branche en branche, ce qui correspond au code successifs des différentes parties de votre compilateur.

On voit aussi qu'il y a 3 "branches de travail" (*working branch* ou *developer branch*) et 2 "branches de rendus" (*release branch*), seules ces dernières sont demandés dans le projet, mais les premières vous seront utiles pour bien séparer votre code "buggué" et votre code "propre".

Exercice 5 (Quelques *commits* pour jouer).
Dans cet exercice, on va jouer à faire faire évoluer nos versions et à naviguer entres elles.

1. Placez-vous dans la branche la plus "primitive", ici `parser_work` :

   ```
   (master) $ git switch parser_work
   ```

   Remarquez que votre prompteur doit avoir changé et indique maintenant que vous suivez `parser_work`.

2. Dans le `README.md`, ajoutez une phrase de description de la branche. Par exemple :

   ```
   Vous vous trouvez sur la branche ``parser_work'' du projet de compilation.
   ********************************************************************************
   Il s'agit d'une branche de travail offrant un exécutable qui analyse lexicalement
   et syntaxiquement le code JS entré et accèpte un code correct dans le fragment
   implémenté.
   ```

3. *commit*ez ce changement :

   ```
   (parser_work) $ git commit -m "desc.  PW" README.md
   ```

   Avec l'option `-m`, on ajoute un message décrivant de changement. [7] Ici on met une description très courte pour pouvoir l'indiquer dans nos schéma, mais il vaut mieux une description un peu plus longue (une petite phrase), et il est généralement conseillé d'y insérer le nom du fichier modifié.

   On ajoute le nom des fichiers modifiés comme arguments de `commit`, on aurait aussi pu faire des `git add` avant, mais procéder ainsi permet d'utiliser la complétion qui n'ajoutera que des fichiers suivis.

4. Vous pouvez vérifier avec un outil graphique (tig, gitk ou "git gui") que l'on a bien deux commits (gitlab a fait un commit initial pour nous) et que +parser_work+ est en avance d'un commit par rapport aux autres branches.

5. On va faire de même avec la branche `parser`. Fermez le README.md (à moins d'utiliser un IDE qui suive les changements de versions). Faites un

   ```
   (master) $ git swich parser
   ```

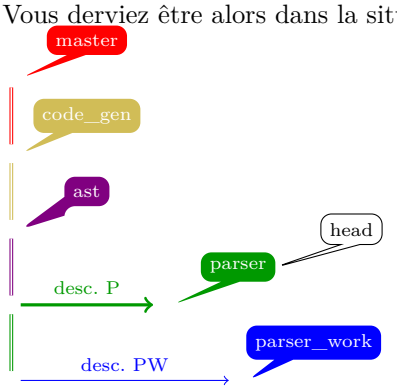
   Et réouvrez le `README.md`. La description ajoutée n'existe plus. C'est normal, on est sur une autre version avant que l'on ai écris cette description. Mais ne vous inquiétez pas, elle existe toujours et vous la retrouverez quand vous re*switch*erez sur la branche `parser_work`.
   Écrivez une description (différente) de la branche `parser` :

   ```
   Vous vous trouvez sur la branche ``parser'' du projet de compilation.
   ********************************************************************************
   Il s'agit d'une branche de Rendu offrant un exécutable qui analyse lexicalement
   et syntaxiquement le code JS entré et accèpte un code correct dans le fragment
   implémenté.
   ```

   Enregistrez vos changements :

   ```
   (parser) $ git commit -m "desc.  P" README.md
   ```

   Vous devriez être alors dans la situation suivante :

   Remarquez que si vous utilisez un outil graphique (gitk, tig ou git gui), vous ne verrez que les deux commits de la branche où vous êtes, car ces outils ne voient que les ancêtres de `Head`.

6. Fermez le README. Revenez dans `parser_work` et vérifiez que vous retrouvez la description écrite au début. La seconde description exite toujours, mais dans la "timeline" de la branche `parser`.

7. Allez sur la branche `parser`. On va y *Merg*-er la branche `parser_work`, cela importe tous les changements effectués dans cette dernière :

7. Sans l'option, il nous serait ensuite demandé la description, car elle est en fait obligatoire.

```
(parser_work) $ git swich parser
(parser) $ git merge parser_work
```

Vous allez avoir une erreur :

```
CONFLIT (contenu) : Conflit de fusion dans README.md
```

Pas de panique c'est normal et arrive souvent lorsque l'on utilise *git* : celui-ci ne sais pas comment traiter les différentes descriptions. En effet, vous avez écrit deux descriptions différentes au même endroit, `git` ne sais pas s'il faut l'une, l'autre, les deux, et dans quel ordre. Il va falloir corriger le merge à la main.

8. Pour ça, allez dans le `README`, au début vous aurez quelque chose de cette forme :

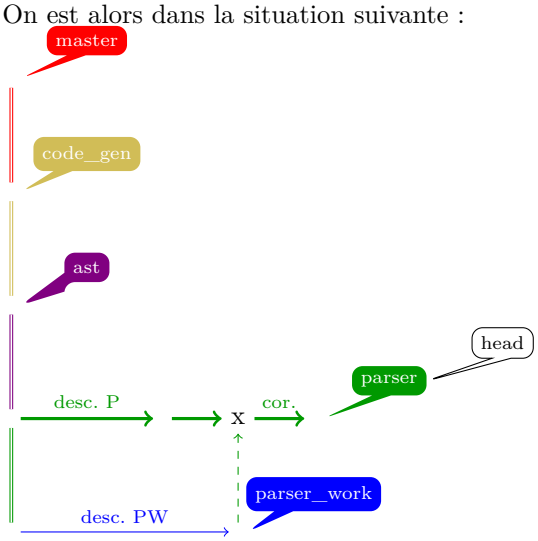
```
<<<<<<< HEAD
Vous vous trouvez sur la branche ``parser'' du projet de compilation.
********************************************************************************
Il s'agit d'une branche de rendu offrant un exécutable qui analyse lexicalement
et syntaxiquement le code JS entré et accèpte un code correct dans le fragment
implémenté.
=======
Vous vous trouvez sur la branche ``parser_work'' du projet de compilation.
********************************************************************************
Il s'agit d'une branche de travail offrant un exécutable qui analyse lexicalement
et syntaxiquement le code JS entré et accèpte un code correct dans le fragment
implémenté.
>>>>>>> parser_work
```

   Cela signifie un conflit entre la version courante, entre `<<<<<<< HEAD` et `=======` et la version *merg*-ée, entre `=======` et `>>>>>>> parser_work` Ici, la description de la branche `parser_work` ne nous intéresse pas, on efface donc la ligne, ainsi que celles avec des `<<<`, `===` et `>>>`.

9. Après ça on doit ré-ajouter le fichier `README.md` (vu que son merge a échoué, il n'est plus vraiment suivi) :

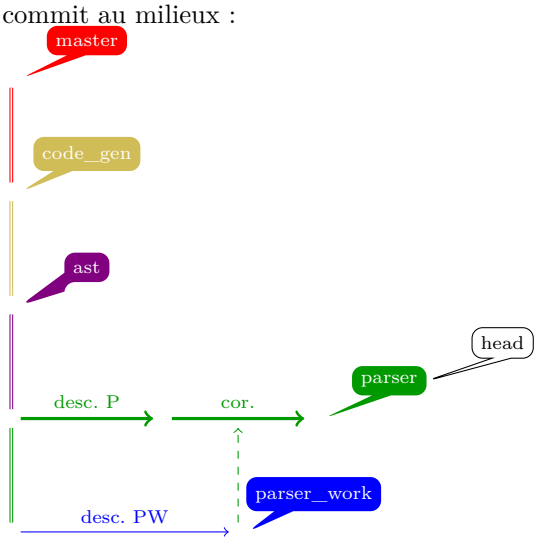
```
(parser) $ git add README.md
(parser) $ git commit -m "cor."
```

   On est alors dans la situation suivante :



   La version avec un petit x est celle avec le conflit. Celle-ci n'existe pas vraiment : il n'y a pas moyen d'y référer et si vous faites un `git log`. Pour `git`, le *commit* du *merge* n'existe pas,

pour bien mettre ce fait en avant, on va plutôt dessiner les deux flèche comme une seule avec le commit au milieux :

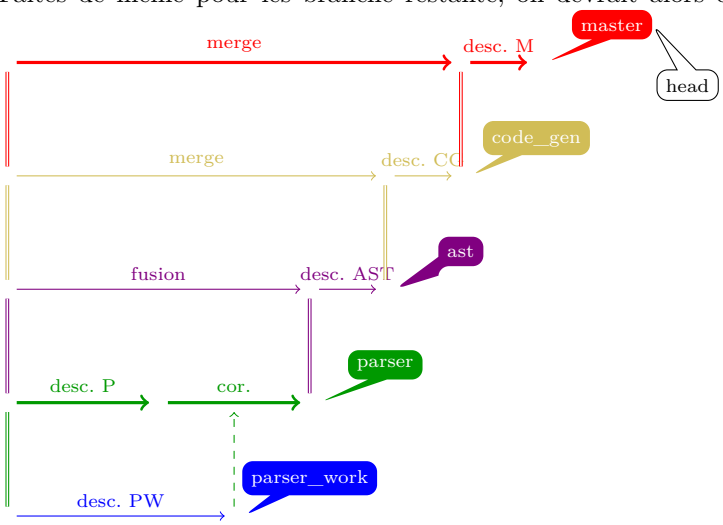


Les outils de visualisation graphique vous donneront maintenant les 4 commits qui sont bien des ancètres de `Head`.

10. Pour éviter ces conflits, il faut au maximum essayer de *merg*-er avant toute modification. En pratique on ne peut pas toujours et il y aura souvent des conflits. Ce n'est pas grave, il faut juste les régler à la main comme on vient de le faire.
11. Placez-vous dans la branche `ast` et faites un merge de `parser` :

    ```
    (parser) $ git switch ast
    (ast) $ git merge -m "fusion" parser
    ```

    Remarquez que l'on a précisé l'option `-m "fusion"` qui donne le nom du commit correspondant au merge (qui a réussi cette fois).
    Cette fusion est aussi un peu particulière : en réalité on n'a pas créé de nouveau commit car la branche `ast` suivait une version qui était une ancètre de celle de `parser`, on a donc juste avancé `ast` vers ce nouveau commit.
12. Modifiez la description [8] et faites un commit.
13. Faites de même pour les branche restante, on devrait alors être dans la situation suivante :



8. N'hésitez pas à demander à un enseignant si vous ne savez pas quoi mettre comme description.

Exercice 6 (Mon premier code).

1. Revenez dans `parser_work`. Créez un fichier `main` dans le langage que vous souhaitez utilisé pour le projet avec un mini programme dedans qui affiche "Entrez un programme JS à parser". Créez aussi un `Makefile`[9] compilant ce dernier. Ajoutez, à la fin du `README.md` la ligne de commande à faire pour lancer le `build` et l'exécution du programme.
2. Pour permettre à git de suivre les fichiers créés, tapez :

   ```
   (parser_work) $ \git{} add Makefile main.c
   ```

   Ici, j'ai supposé que l'on avait un makefile et que l'on était en *C*. Évidement vous devez remplacer les noms de fichiers si besoin.
   Remarque : N'UTILISEZ JAMAIS la commande "`git add .`", celle-ci ajouterait l'exécutable, or on ne doit jamais versionner des fichier générés. [10]
   Faites un *commit* :

   ```
   (parser_work) $ git add README.md
   (parser_work) $ git commit -m "1er prog."
   ```

   Remarquez que je ne précise pas que `Makefile` et `main.c` doivent être ajoutés au *commit* : c'est automatique après le `add`.
3. Allez sur la branche `parser`. Les fichiers créés ont disparu mais pas l'exécutable, encore une fois c'est normal, ils sont enregistrés mais n'existent pas dans cette "timeline" ; quand à l'exécutable, il n'est pas suivi par `git`, il est donc ignoré par toutes es commandes `git`. Celà peut être un soucis pour le makefile qui est des fois perdu dans les version, il est plus propre de précéder les `git switch` par un `make clean`.
   On va y *Merg*-er y la branche `parser_work`, cela importe tous les changements effectués dans cette dernière :

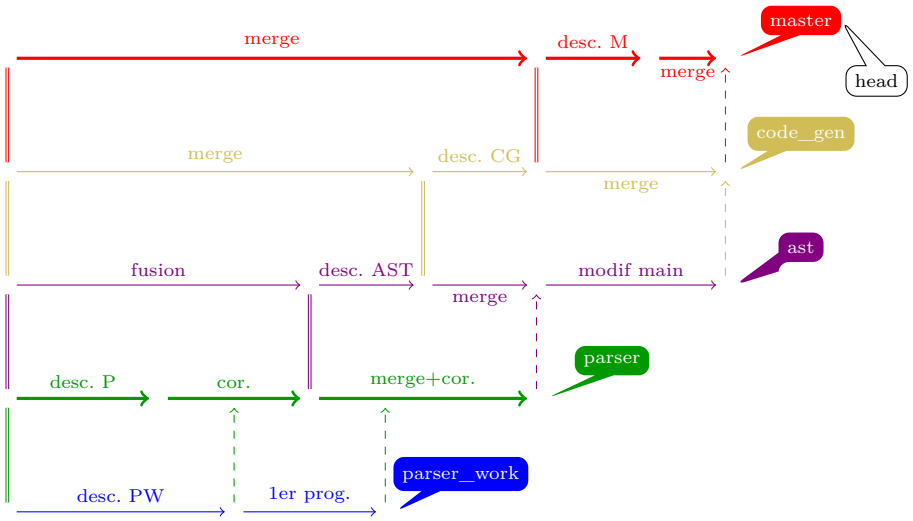
   ```
   (parser_work) $ git swich parser
   (parser) $ git merge parser_work
   ```

   Vous allez avoir une erreur :

   ```
   CONFLIT (contenu) : Conflit de fusion dans README.md
   ```

   Encore une fois, c'est normal : `git` arrive à créer le makefile et le main (vérifiez avec `ls`), mais il n'arrive pas à ajouter la ligne de *build* dans le `README.md` car les deux `README.md` sont trop différents. Lorsque vous avez un gros conflit, cela en créera d'autres après, d'où l'intérêt de faire des *merge* réguliers.
4. Faites propagez aux autres branches sans créer de conflit et en changeant la phrase affichée lorsque l'on commence les étapes de compilation.
   Vous devriez avoir le *workflow* suivant :

9. ou `run.sh`, ou `dune` ou n'importe quel autre fichier de *build* que vous avez l'habitude d'utiliser
10. En réalité, on peut l'utiliser si on maintient un `.gitignore` complet, ce que est difficile.



Exercice 7 (Pousser sur le dépôt). Il n'y a, pour l'instant, rien d'enregistré dans le dépôt. Si vous avez un binôme qui a accès au dépôt, il ne verra rien de tout ce que vous avez fait.

1. Placez-vous dans la branche `parser_work` et faites la commande

   ```
   (parser_work) $ git push
   ```

   Vous devriez voir un message d'erreur. C'est normal : vous poussez sur une branche que le serveur ne connais pas. Il faut lui indiquer que vous voulez créer cette branche au sein de celui-ci. Heureusement, le message d'erreur vous donne la commande à taper.
2. Poussez ainsi chacune des branches.
3. Si vous êtes en binôme, demandez à votre binôme de reprendre la main en faisant un clone du dépôt sur son ordinateur et en faisant un `pull`. Il devrait alors avoir accès à tout ce que vous avez fait, y compris l'historique des *commits* (visibles avec gitk ou tig).
4. Si votre binôme est d'accord avec cette version, il peut placer le *tag* sur les branches de rendu :

   ```
   (......) $ git switch parser
   (parser) $ git tag "p"
   (parser) $ git switch master
   (master) $ git tag "c"
   ```

   Le tag donne un nom et fige définitivement un commit (on peut néanmoins supprimer un tag si on a fait une bêtise). C'est important car il est possible de réorganiser les autres commits après coup [11], or un utilisateur doit toujours avoir accès à une version spécifique non modifiée (corriger un bug peut en créer un autre...).
5. Pousser un tag sur le serveur est un peu bizarre car 1) les tag sont arrivés plus tard, et 2) vous pouvez vouloir avoir des tags en local qui ne sont pas poussés.
   Pour ça tapez :

   ```
   (master) & git push origin c
   (master) $ git switch parser
   (parser) & git push origin p
   ```

   Ici, "origin" désigne le serveur utilisé (on peut avoir plusieurs serveurs de dépôt), et "c" et "p" désignent les tags poussez.
   Vous pouvez vérifier qu'ils sont visible sur le gitlab. Si votre binôme fait un `git pull`, il doit pouvoir les voir aussi avec la commande `git tag`.

11. et c'est souvent fait, car l'historique d'un projet a aussi une valeur de documentation il faut donc qu'elle soit propre

Les tags suivent des conventions standard : ils sont de la forme `b.x.y` ou `b.x.y.z`

— Le `b` est le nom ou un diminutif de la branche de *release* utilisé (chez nous `c` pour la branche `master` et `p` pour la branche `parser`).
— Le `x` est le numéro de version (correspond dans le projet à la notion de gros fragment), un changement indique une version radicalement différente.
— Le `y` est le numéro de sous-version (correspond dans le projet à la notion de fragment).
— Le `z` est utilisé lorsque l'on a besoin de corriger une sous-version déjà *tag*gée (Pas demandé dans le projet, mais a utiliser si vous trouvez des bogs après coup).

TP1 : Vous pouvez commencer le TP1, vous allez devoir appliquer le *workflow* vu ici en ajoutant du vrais contenu dans chaque branche. Veuillez prendre soin de bien respecter quelques pratiques :

— VFaites bien attention à la branche sur laquelle vous êtes avant d'ouvrir un fichier.
— Si vous avez l'ipression que queque chose a disparu, ne faites pas un copié-collé, mais cherchez plutôt la raison de la disparition (bien souvent vous êtes simplement sur la mauvaise branche).
— Avant de changer de branche, fermez toujours les fichiers ouverts [12]
— Ne suivez JAMAIS de fichiers générés via `git`. Pour ça, préciser toujours quel fichier ajouter en utilisant `git add` en n'ajoutant que les fichiers que vous avez écrit vous même.
— Pensez à faire un `make clean` avant de changer de branche (moins critique mais plus propre).

***Workflow* :** Ce *workflow* avec des branches en parallèles est très commun. Il est aussi très utilisé lorsque l'on est sur des gros projets avec des versions *alpha*, *beta*, *testing*, etc... Mais ce n'est pas le seul workflow possible, et il est même possible de combiner des workflow. Voici quelques exemples :

— Sur les projets à moins de 10 mais qui sont assez critique, les développeurs sont souvent appelés à avoir chacun leur branche, et un membre est chargé de faire les *merge*s dans la branche master (ou dans une branche de merge pour maintenir master propre).
— Pour faire des tests un peu complexe, on peut dupliquer certaines branches avec une version de tests depuis laquelle on merge la version principale mais de laquelle on ne revient jamais, cela permet d'ajouter du code de test au milieu du projet (des *prints* et des *asserts* notamment) sans polluer ce dernier. Dans le TP2, nous vous proposons d'en créer une, mais ce n'est pas obligatoire.
— Des branches exploratoires qui permettent d'essayer des choses "pour voir", on en fera une dans le TP1. Celles-ci sont amenées, à terme, à disparaître, mais on peut les garder au cas où (c'est toujours du code, pourquoi le jeter ?).
— ...

12. Beaucoup d'IDE peuvent s'en sortir sans fermer les fichiers, mais il faut les paramétrer pour ça.