TP Git
(préalable au cours de compil)

15 août 2022

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. 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" : faites le chez vous avant le premier TP et posez nous les questions en début d'heure.

Exercice 1 (cours et TP de Thierry Monteil). Si vous n'êtes pas à l'aise du tout avec Git, vous pouvez regarder les *slides* de T. Monteil (niveau L2) et commencer par le TP associé sections 2 à 10. Ceux-ci vos permettront de vous familiariser avec les manipulation purement locales et mono-branches de GIT. On y voit les commandes `git add`, `git commit`, `git push`, `git status` et `tig`.

Si vous êtes perdu dans les branchements, utilisez l'interface graphique (`gitk` ou `tig`), ou encore un simulateur.

Exercice 2 (Mise en place d'un dépot git). Avant tout, je dois corriger 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) implémentant ce protocole.

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 hébèrgeur comme github.

1. Allez sur https://gitlab.sorbonne-paris-nord.fr et identifiez vous via l'option **LDAP** avec vos identifiants 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 `Edite 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 M1". Pour la durée du semestre, laissez privé affin d'éviter les fuites... [1]
4. Allez sur la page du projet nouvellement créé. Cliquez sur **Clone** et faites une copie du premier lien. [2] Utilisez votre invité de commande en vous plaçant dans le dossier où vous souhaitez

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...

2. Si vous utilisez *visual-studio* ou *InteliJ*, vous pouvez aussi l'importer via votre IDE, mais les lignes de commandes décrites pendant ce TP seront à remplacer par des manipulations dans le menu de votre IDE.

enraciner votre projet et tapez :

```
$ git clone lien_que_vous_venez_de_copier
```

5. Il vous reste à inviter votre binôme et/ou vos enseignants : 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 [3] de projet comme `Developer` ou `Maintainer` et chacun de vos enseignants comme `Repporter`.

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

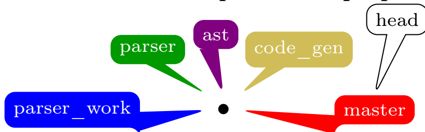
Attention : Une erreur courante sur ce TP consiste à remplacer brutalement un fichier par la correction ou par la version présente sur une autre branche git. Il ne faut jamais faire ça, car l'algorithme de différenciation de git n'arrivera pas à suivre et les future *merge* risquent de tous échouer. 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. [4]

Exercice 3 (Mise en place du "*workflow*").
Au début du TP, vous devez vous placer dans le dossier cloné, vous serez automatiquement sur la branche `master`, [5] 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` :

```
$ git branch parser_work
$ git branch parser
$ git branch ast
$ git branch code_gen
```

Cela créer 4 pointeurs qui pointent la même version :



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

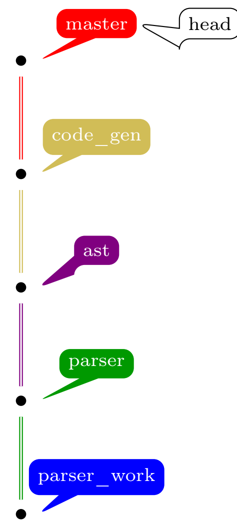
Notre dessin n'est pas très lisible car on ne comprends pas l'intérêt d'avoir plein de branche, on va

---

3. 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.

4. 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.

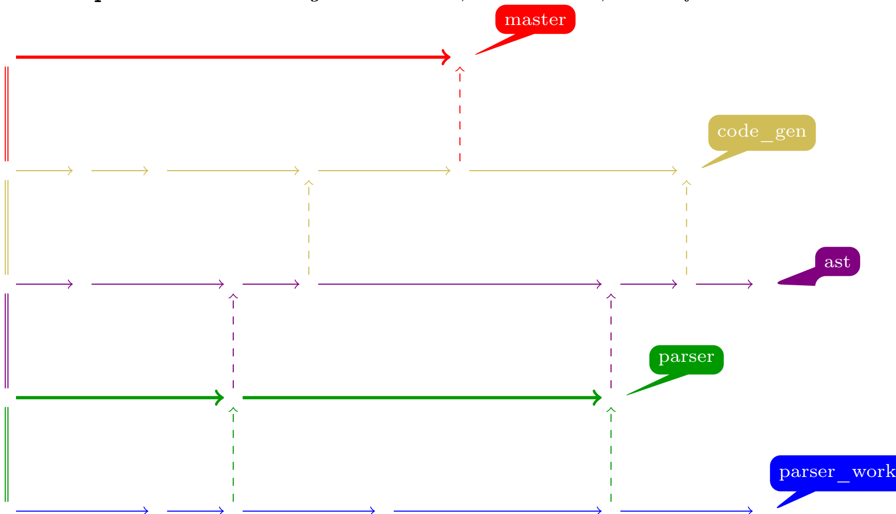
5. Selon le dépôt git utilisé, votre branche `master` s'appellera `main`, dans ce cas remplacez simplement "master" par "main" dans toutes les commandes données.



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'ils soient marqués au dessus nous permettent de dire qu'elles sont "moralement plus avancées".

En effet, tous les changements faites des `parser_work` seront à un moment *mergé*s dans `parser`, ceux de `parser` seront *mergé*s dans `ast`, *etc*... Ainsi, notre *flow* devra ressembler 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 "branche de travail" (*working* ou *developer*) et 2 "branches de rendus" (*release*), seules ces dernières sont demandés dans le projet, mais les premières vous seront utiles pour bien séparer votre code.

Exercice 4 (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` :

   ```
   $ git swich parser_work
   ```

2. Dans le `README`, 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 bran
   ```

3. *commit*ez ce changement :

   ```
   $ git commit -m "desc. br. PW" README
   ```

   Avec l'option `-m`, on ajoute un message décrivant de changement. [6] 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.

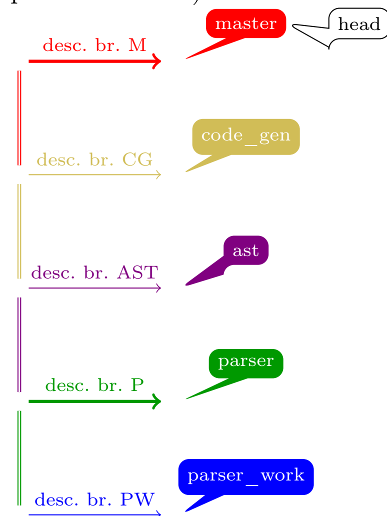
   On ajoute le nom des fichiers modifiés comme arguments de `commit`, on aurait aussi pu faire des `git add` avant, mais procéder ainsi permet d'utiliser la complétion qui n'ajoutera que des fichiers suivis.

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

   ```
   $ git swich parser
   ```

   Et réouvrez le `README`. 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 de la branche `parser` et *commit*ez.

5. Faites de même avec chaque branche (un `switch` puis l'ajout de la description dans le `README`, puis un commit). On devrait alors être dans la situation suivante :



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

6. Revenez dans `parser_work` et vérifiez que vous retrouvez la description écrite au début.
7. Créez un fichier `main` dans le langage que vous souhaitez utiliser pour le projet avec un mini programme dedans (un hello-world par exemple). Créez aussi un `Makefile` [7] compilant ce dernier. Ajoutez, à la fin du `README` la ligne de commande à faire pour lancer le `build` et l'exécution du programme.
8. Pour permettre à git de suivre les fichiers créés, tapez :

   ```
   $ 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.
   Faites un *commit* :

   ```
   $ git commit -m "w1er prog." README
   ```

   Remarquez que je ne précise pas que `Makefile` et `main.c` doivent être ajoutés au *commit* : c'est automatique après le `add`.
9. Allez sur la branche `parser`. Les fichiers créés ont disparu, encore une fois c'est normal, ils sont enregistrés. et *merg*-ez y la branche `parser_work` :

   ```
   $ git swich parser
   $ git merge parser_work
   ```

   Vous allez avoir une erreur. Pas de panique c'est normal : git arrive à créer le makefile et le main, il arrive aussi à ajouter la ligne de *build* dans le `README`, mais il ne sais pas comment traiter les 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.
10. Pour ça, allez dans le `README`, au début vous aurez quelque chose de cette forme :

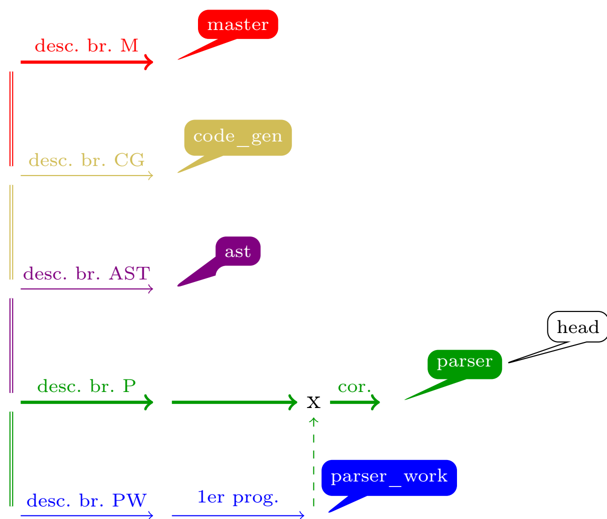
    ```
    <<<<<<< HEAD
    description_de_la_branche_parser
    =======
    description_de_la_branche_parser_work
    >>>>>>> parser_work
    description_du_dépot
    ligne_de_build
    ligne_de_run
    ```

    Ici, la description de la branche `parser_work` ne nous intéresse pas, on efface donc la ligne, ainsi que celles avec des `<<<`, `===` et `>>>`.
11. Après ça on doit ré-ajouter le fichier `README` (vu que son merge a échoué, il n'est plus vraiment suivi) :

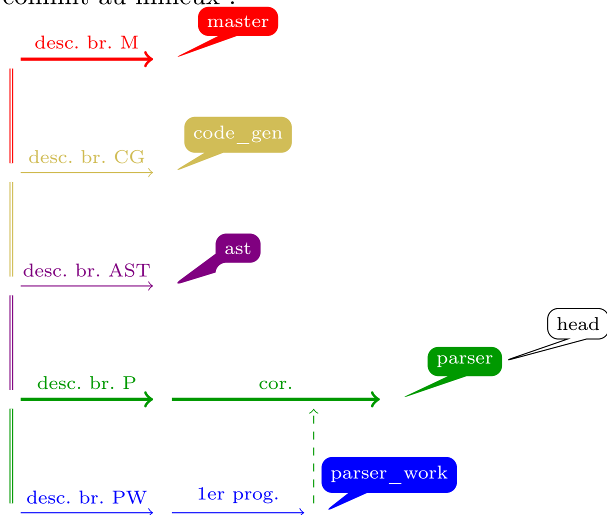
    ```
    $ git add README
    $ git commit -m "cor."
    ```

    On est alors dans la situation suivante :

7. ou `run.sh`, ou `dune` ou n'importe quel autre fichier de *build* que vous avez l'habitude d'utiliser



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 :

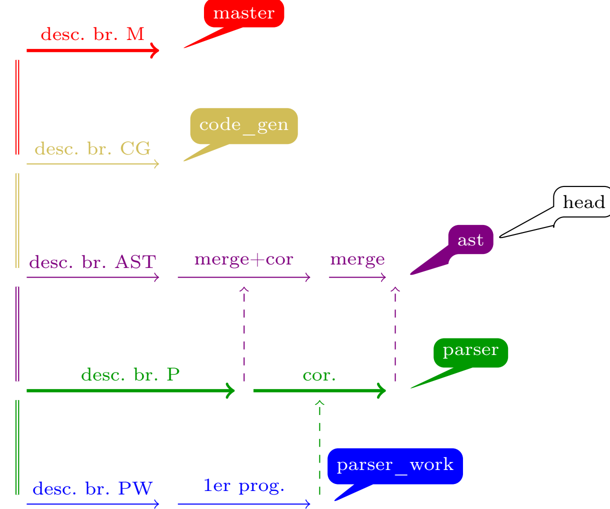


12. On voudrait faire pareil avec les autres, mais vu que l'on sais maintenant que l'on va avoir un conflit entre les description, on voudrait résoudre le conflit avant de faire le merge complet. Pour ça, on va faire le merge en deux fois : d'abord *mergez* non pas `parser`, mais la précédente version valide de `parser` (indiquée avec le `^`),[8] puis corrigez et re*mergez* :

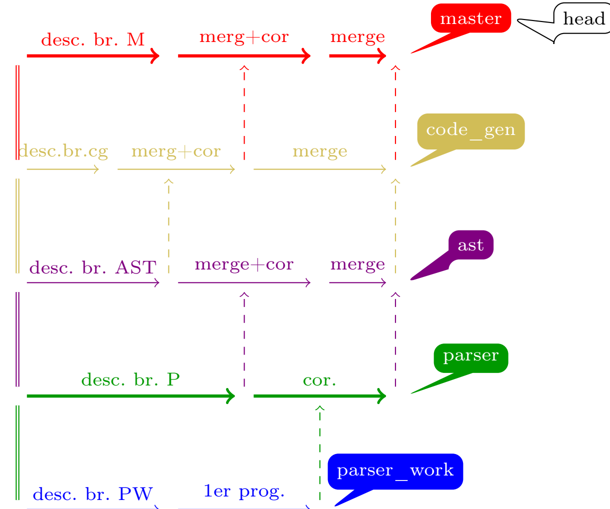
```
$ git switch ast
$ git merge parser^
... conflit ...
$ nano README
... résolution à la main ...
$ git add README
$ git commit -m ‘‘merge+cor’’
$ git merge parser
... pas de conflit ...
```

8. Sur Windows, le ^ ne fonctionne pas, utilisez `^1`

Cela correspond au schéma :



13. En utilisant l'une des deux méthodes vues (ou en faisant le premier merge après `merge+cor`), faites de même pour les branches `code_gen` et `master`.
Remarque : si vous avez besoin de revenir de deux commits valables depuis `ast`, il faut utiliser `ast^^`.
Par exemple, on peut avoir l'historique suivant :



14. Vous pouvez visualiser cet historique avec la commande `tig` ou l'aperçu en haut à gauche de la fenêtre de `gitk` (ou un autre outils graphique si vous en avez). L'historique que vous aurez est moins lisible car 1. les commit initiaux sont fondus en un (on avait triché en les séparant), 2. la structure en branche parallèle est moins visible (git ne connaît pas votre workflow), et 3. les *merge*s réussis ou avec corrections sont indiqués de la même manière.
15. Placez-vous sur une autre branche (avec `git switch`) et réaffichez l'historique. Que remarquez-vous ?

TP1 : Dans le TP1, vous allez pouvoir appliquer ça en ajoutant du vrais contenu dans chaque branche. Faites bien attention à la branche sur laquelle vous êtes (`git branch` ou `git status`).

***Tags* :** Il y a un aspect de GIT essentiel pour le projet que l'on n'a pas vu dans ce TP0 : les *tags*. Ceux-ci permettent de fixer de manière *immutable* des versions avec la date et un *tag*. C'est important car il est possible de réorganiser les autres commits après coup (et c'est souvent fait, car l'historique d'un projet a aussi une valeur de documentation il faut donc qu'elle soit propre). Ce n'est pas un aspect que l'on verra mais il est important de *tag*ger les versions importantes. 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).

***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 ?).
— ...