Voici un **plan de route** détaillé pour avancer à deux sur votre Minishell à partir du squelette actuel. L’objectif est de montrer **quoi faire**, **dans quel ordre**, et **comment se synchroniser** efficacement. Nous allons vous proposer des **étapes** et préciser, pour chacune, **qui** fait quoi (Étudiant A ou Étudiant B). Vous pourrez ajuster selon vos préférences, mais l’idée est de vous donner un cadre clair.

**1. Rappel du squelette actuel**

* **main()** : contient la boucle readline(), gère Ctrl+D, a un shell\_env passé dans la boucle.
* **signal\_setup** : installe un handler qui met à jour g\_received\_signal.
* **create\_shell\_env / destroy\_shell\_env** : gèrent la copie de l’env et l’allocation dynamique.
* **Une seule variable globale** g\_received\_signal.

Tout le reste (parsing/exécution/builtins/expansions) est à faire.

**2. Étapes et répartition des tâches**

**Étape 1 : Préparer la base pour le parsing**

**Étudiant A (Parsing)**

1. **Définir la structure de donnée pour stocker la commande**
   * Ex. t\_cmd ou t\_cmd\_list (liste chaînée ou tableau).
   * Chaque élément contient :
     + char \*\*argv (liste d’arguments de la commande),
     + Informations de redirection (fichier in/out, <<, >>, etc.),
     + Pointeur vers la commande suivante si un | existe.
2. **Créer une fonction parse\_command\_line()** qui :
   * Découpe la ligne en tokens (attention aux quotes).
   * Détecte et enregistre les redirections (<, >, <<, >>).
   * Gère le pipe | pour séparer les commandes.
   * Retourne une liste de t\_cmd (chaînée ou un tableau).

Pendant ce temps…

**Étudiant B (Exécution / Environnement)**

1. **Préparer un module “execution”** avec :
   * Un prototype execute\_commands(t\_cmd \*cmds, t\_shell\_env \*shell\_env).
   * À ce stade, ça peut juste afficher les commandes reçues (« Debug print ») pour valider le parsing.
   * Mettre en place une fonction setup\_redirects() vide ou semi-fictive pour plus tard, quand A aura défini la structure.
2. **Gérer l’environnement en interne**
   * Mettre à disposition des fonctions style env\_get(var), env\_set(var, value), env\_unset(var).
   * Ceci prépare le terrain pour les builtins export/unset.

**Synchronisation sur l’étape 1**

* **Étudiant A** donne à **B** un fichier d’en-tête (par ex. parser.h ou un simple minishell.h) décrivant la structure t\_cmd.
* **Étudiant B** appelle parse\_command\_line() dans le main, récupère la structure de commandes, et fait un **simple debug** (print).

Objectif : **valider** que la structure de parsing est lisible et exploitable.

**Étape 2 : Intégrer la gestion des builtins de base**

**Étudiant B** :

1. **Créer une table de dispatch** des builtins. Exemple :
2. typedef struct s\_builtin {
3. char \*name;
4. int (\*func)(char \*\*argv, t\_shell\_env \*shell\_env);
5. } t\_builtin;
6. t\_builtin g\_builtins[] = {
7. {"cd", &builtin\_cd},
8. {"exit", &builtin\_exit},
9. {"pwd", &builtin\_pwd},
10. {"echo", &builtin\_echo},
11. {"env", &builtin\_env},
12. {"export", &builtin\_export},
13. {"unset", &builtin\_unset},
14. {NULL, NULL}
15. };
16. **Implémenter** au moins quelques builtins (au choix) :
    * echo (avec option -n),
    * pwd,
    * exit (pour tester la sortie).
17. **Créer une fonction** int is\_builtin(char \*cmd\_name) et int exec\_builtin(t\_cmd \*cmd, t\_shell\_env \*shell\_env) qui :
    * Compare le cmd->argv[0] au tableau de builtins,
    * Si trouvé, appelle la fonction correspondante,
    * Sinon, retourne 0 (pas un builtin).

**Étudiant A** :

* Continue d’améliorer le parsing si besoin (gestion des quotes correctement, etc.).
* Gère la **découpe en argv** pour chaque commande.
* Peut s’occuper de la **partie expansions** (si déjà validé) ou au moins commencer à l’implémenter pour $VAR, $?.

**Synchronisation sur l’étape 2**

* **A** doit s’assurer que t\_cmd->argv[0] contient bien la commande (ex: ls, echo, etc.).
* **B** peut appeler if (is\_builtin(cmd->argv[0])) exec\_builtin(...); else exec\_external(...).

Objectif : vérifier que certains builtins (ex: echo, pwd) sont fonctionnels même sans redirection, sans pipe.

**Étape 3 : Implémenter l’exécution des commandes externes**

**Étudiant B (Exécution)**

1. **Créer exec\_external()** :
   * Gérer la recherche dans le PATH, par ex. :
   * char \*resolve\_path(char \*cmd, char \*\*env);
   * // Renvoie "/bin/ls" si cmd = "ls", ou cmd si c'est un chemin absolu
   * Faire un fork(), dans le fils → execve(path\_resolved, cmd->argv, shell\_env->env).
   * Récupérer le code retour via waitpid() et l’enregistrer dans shell\_env->exit\_status.
2. **Gérer les erreurs** (commande introuvable, permission denied, etc.).

Pendant ce temps…

**Étudiant A (Parsing)**

* Peut commencer à **gérer les expansions** ($VAR, $?) au moment où il découpe la ligne.
* **Ne pas** oublier que les single quotes ' ' **désactivent** l’expansion, alors que les double quotes " " l’autorisent.
* Ajouter éventuellement la gestion **basique** des redirections (repérer > file, < file, etc.) dans la structure t\_cmd.

**Synchronisation sur l’étape 3**

* **B** aura besoin d’avoir cmd->infile, cmd->outfile (ou un tableau de redirections) pour plus tard.
* **A** peut livrer un t\_cmd où infile et outfile sont initialisés.
* **B** pour l’instant peut ignorer les redirections (ou stub) et juste exécuter la commande.
* **Test** : taper ls, ./a.out, etc. dans votre shell, voir si ça fonctionne.

**Étape 4 : Gérer redirections et pipes**

**Étudiant A** :

1. Dans parse\_command\_line(), détecter :
   * > = redirection sortie (truncate)
   * >> = redirection sortie (append)
   * < = redirection entrée (fichier)
   * << = here\_doc (lecture jusqu’à un délimiteur)
2. Stocker ces infos dans t\_cmd (ex. t\_redir type; char \*filename;).
3. Si plusieurs commandes séparées par |, créer autant de t\_cmd chainés (ou un tableau).

**Étudiant B** :

1. **Implémenter setup\_redirects(cmd)**
   * Selon le type, ouvrir le fichier, faire dup2() sur STDOUT\_FILENO ou STDIN\_FILENO.
   * Gérer le cas << (ici-doc) : lire la saisie jusqu’au délimiteur (ou la stocker dans un pipe, etc.).
2. **Implémenter execute\_pipeline(cmd\_list, shell\_env)** pour enchaîner les fork() + pipe().
   * Par exemple, si on a cmd1 | cmd2 | cmd3, on crée 2 pipes, on exécute cmd1 dans un fils, cmd2 dans un autre, etc.

**Synchronisation sur l’étape 4**

* **A** doit donner à **B** une structure claire pour les redirections (type, filename).
* **B** appelle setup\_redirects avant l’exec (dans le fils).
* Testez des commandes comme ls > out.txt, cat < file, cat file | grep hello.

**Étape 5 : Finaliser les builtins et la gestion de l’environnement**

**Étudiant B** :

* Implémente tous les builtins restants :
  + env : affiche shell\_env->env,
  + export : insère/modifie une variable,
  + unset : supprime la variable,
  + cd : gère le changement de répertoire,
  + exit : met shell\_env->running = 0 et éventuellement exit\_status.

**Étudiant A** :

* Continue à peaufiner le **parsing des expansions** (cas $VAR inexistant, $?, etc.).
* Vérifie que export, unset ont un effet si on retape une commande qui exploite la variable ajoutée/supprimée.

**Synchronisation sur l’étape 5**

* **A** et **B** se mettent d’accord sur la forme de shell\_env->env. B en aura besoin pour manipuler/exporter.
* **Test** : export MYVAR=hello, puis echo $MYVAR. Ensuite unset MYVAR, echo $MYVAR.

**Étape 6 : Gérer les signaux restants et le code de retour**

**Étudiant B** :

* Vérifier SIGQUIT (Ctrl+) pour les programmes en cours d’exécution (ex: cat).
  + Si un process fils est en cat, Ctrl+\ doit lui envoyer SIGQUIT. Dans votre shell, vous ignorez peut-être ou vous gérez différemment.
* Vérifier que g\_received\_signal est remis à 0 à chaque tour, et ajuster exit\_status comme bash (130 pour SIGINT, 131 pour SIGQUIT, etc.).

**Étudiant A** :

* Tester tout particulièrement le comportement dans le parsing quand on tape un Ctrl+C **pendant** qu’un here\_doc attend une entrée, etc.
  + Il se peut que vous deviez interrompre la lecture du here\_doc quand SIGINT arrive.

**Synchronisation sur l’étape 6**

* Communiquer sur la gestion de signaux en cours d’exécution vs. en attente de commande.
* Contrôler que le exit\_status se met bien à jour dans shell\_env.

**Étape 7 : Nettoyage, tests et corrections**

* **Tous les deux** :
  1. **Tester systématiquement** toutes les fonctionnalités (pipelines complexes, redirections multiples, expansions, etc.).
  2. **Corriger** les fuites mémoire (tester avec valgrind).
  3. **Respect de la Norme** (nommage, fonctions < 25 lignes, etc.).
  4. Éventuellement, commencer les **bonus** (&&, ||, wildcards \*, etc.) si le mandatory est **parfait**.

**3. En résumé : tableau de répartition**

| **Étape** | **Étudiant A** | **Étudiant B** | **Synchronisation clé** |
| --- | --- | --- | --- |
| **1. Parsing (base)** | - Définir structure t\_cmd - parse\_command\_line() minimal (split) | - Créer execute\_commands() squelette - Fct. d’environnement (env\_get/set) | A -> donne à B la structure de commande. B -> debug print. |
| **2. Builtins (début)** | - Améliorer parsing (quotes, expansions basiques) | - Table de dispatch g\_builtins[] - Implémenter echo, pwd, exit | Vérifier que cmd->argv[0] arrive intact pour repérer la builtin. |
| **3. Exécution externe** | - Gérer expansions plus fines ( $VAR, $? ), single quote vs double quotes | - exec\_external(): recherche PATH, fork(), execve() - Gérer erreurs | A -> expansions correctes B -> exécuter la commande si pas builtin. |
| **4. Redirections / Pipes** | - Parsing : détecter <, >, <<, >>, ` | <br/> - Stocker infos danst\_cmd` | - setup\_redirects() - execute\_pipeline() (plusieurs fork et pipe()) |
| **5. Builtins avancées** | - Affiner expansions / parsing (cas spéciaux) | - Implémenter env, export, unset, cd - Interaction avec shell\_env->env | Tester export MYVAR=... et echo $MYVAR, etc. |
| **6. Signaux et code retour** | - Gérer Ctrl+C pendant un here\_doc - Vérifier parsing interrompu | - Ajuster g\_received\_signal pour SIGINT, SIGQUIT - Mettre exit\_status correct | Contrôler le comportement si on interrompt un programme ou un here\_doc. |
| **7. Finalisation / Tests** | - Tester, debugger, corriger - Respect norme, no leaks | - Tester, debugger, corriger - Respect norme, no leaks | Validation conjointe du mandatory, envisager le bonus ensuite. |

**4. Conseils de collaboration au quotidien**

1. **Pull Requests** ou **Merge Requests** :
   * Quand l’un finit une étape, l’autre la revoit avant de “merger”.
2. **Points de synchro** :
   * Brief quotidien sur ce qui a été fait, ce qui reste à faire, blocages.
3. **Tests unitaires** :
   * Dès qu’A a fini un parsing, B fait un mini test.
   * Dès que B fait un builtin, A teste, etc.

En suivant ce plan, vous saurez **qui fait quoi** et **dans quel ordre**. L’important est de **valider à chaque étape** pour éviter d’avoir à tout refondre plus tard.

Bon courage pour la suite !