Notre projet est en Ocaml, le lexer utilise Ocamllex et le parser Menhir.
Le plus compliqué et long à faire fut de loin le typage, il est désormais correcte sur 100% des tests fournis. J'ai abondé mon code de commentaires, en expliquant ce que j'ai compris de java, les problèmes que j'ai eu et surtout les solutions apportées. Partant d'une méconnaissance totale des langages objets, la première difficulté fut d'apprendre les bases de Java (je remercie Félix et Naëla), mais aussi les comportements plus spécifiques voire parfois obscure de Java. En parallèle de mon avancé dans le projet, je testais le comportement de java et j'ai laissé mon dossier tests_perso au complet car j'y fais référence dans mes commentaires (même si de nombreux tests furent avortés et je n'y fais plus référence). Brièvement mon code s'articule ainsi :

- Je prépare le terrain en déclarant mes variables globales.
- Je récupère le nom des classes et des interfaces, et je place des informations utiles dans les variables globales.
- Je déclare les liens d'héritages entre les classes et entre les interfaces.
- Je fais un tri topologique pour m'assurer qu'il n'y a pas de cycle dans les héritages et pour plus tard déclarer les champs et les méthodes dans un ordre topologique.
- Pour chaque classe/interface je déclare ses types paramètres, en cherchant là aussi des ordres topologiques.
- Viennent ensuite toutes les fonctions auxiliaires utiles, d'abord : la gestion des substitutions des paramètres types, les vérifications d'extensions, d'implémentations, de bien fondé et de sous-type.
- Suivent d'autres fonctions pour la gestion des méthodes : en hériter, en déclarer de nouvelles.
- Je vérifie le bon typage des interfaces en suivant l'ordre topologique et à l'aide des fonctions précédentes. J'en profite pour déclarer les méthodes.
- Idem pour les classes, où je déclare les méthodes, les champs et le constructeur.
- Seulement après, pour toutes les classes, je déclare les méthodes et les champs des types paramètres.
- Je m'attaque ensuite aux corps, à nouveau en commençant par des fonctions auxiliaires, qui vérifie le typage des accès, des expressions et des expr simples, puis des instructions.
- Enfin je vérifie le corps de toutes les méthodes et tous les constructeurs, en terminant avec l'unique méthode de Main.

Remarque : ne sachant pas à l'avance exactement ce qui nous servira pour la production de code, pour le moment ma fonction ne produit rien. Elle renvoie une erreur si il y en a une, sinon elle dit juste que le typage s'est bien passé. Je préférais ne pas m'encombrer avec ça, au vu du nombre de modifications que je devais régulièrement faire pour aboutir à quelque chose de correcte et pratique (passer de Hashtbl aux Map ou l'inverse, ou changer le type de retour d'une fonction par exemple).

La liste de mes difficultés rencontrées est longue et les brouillons se sont accumulés sur mon bureau, mais les commentaires dans le programme retranscrivent déjà tout ce que j'ai jugé intéressant. Brièvement :

- Comme je le dis au tout début de mon programme, je garde les informations à trois endroits différents :
  - Les variables globales pour les informations très générales concernant les "vraies" classes et interfaces (comprendre par là, pas les types paramètres).
  - Un environnement de typage global : env_typage_global, qui constitue tout ce qui est commun à tous les environnements locaux. Par exemple env_typage_global.methodes est une Hashtbl qui pour un nom de classe/interface, renvoie l'ensemble de ses méthodes (une methtab).

— Les env_typage locaux aux vraies classes et interfaces, avec en plus par rapport au global, les informations concernant les types paramètres. Par exemple on trouve dans env_typage.methodes les méthodes de chaque types paramètres, de même pour les .extends .implements ou .champs.

J'ai trouvé ça très pratique pour ne pas mélanger les informations, en revanche ça été source d'erreurs d'inattentions lors des tests. En effet il faut être vigilant à dans quel env_typage sont les informations. Si dans le corps d'une méthode d'une classe A on fait référence à une méthode de la classe B ayant pour type de retour T un paramètre de type, il faut penser à le substituer. De même lorsqu'on hérite de champs ou de méthodes. Je m'y suis pris à plusieurs fois avant d'atteindre cet équilibre entre ce que je mets dans les variables globales et dans les env_typages. J'en suis plutôt satisfait, car je le trouve pratique, même si ce n'était pas probablement pas la solution attendue. Pour la vérification des corps j'utilise des env_vars pour retenir les variables locales, avec leurs types et si elles sont initialisées.

— Comme je le dis en introduction, j'ai pris du temps pour cerner les comportements de java. Petit exemple : si une interface hérite d'une méthode via deux interfaces mères, alors il faut que les deux types de retours soient en relation, l'un doit être sous type de l'autre, et dans ce cas on considère uniquement le type le plus petit.

J'ai surtout eu de sérieuses difficultés à gérer l'initialisation des variables locales. Le fait de pouvoir dire `I c;` induit des vérifications pour savoir si oui ou non `c` est désormais initialisée. Car il faut planter si on utilise `c` sans être sûr de l'avoir initialisée. Par exemple `I c; if (b) {c = new C(); } c.m();` est incorrecte (SAUF avec `if (true)`). Encore une fois, j'espère que mon code avec les commentaires est parfaitement compréhensible, tous les détails y sont.

— Dans le même genre, j'ai du m'y reprendre à plusieurs fois avant de gérer les returns proprement.

— Cerner les différences entre les "vraies" classe/interface et les types paramètres est crucial. Globalement les types paramètres ont un comportement proche des classes, mais typiquement parler d'un constructeur n'a pas de sens pour un type paramètre. Avec ma solution cela ne pose pas de grande difficultés, mais de nombreuses petites erreurs d'inattentions.

— Autre petit comportement d'apparence obscure pour un néophyte en programmation objet, un type paramètre peut en étendre un autre MAIS dans ce cas ce doit être sa seule contrainte : `C<Y extends X,X extends A&I,Z extends J>` est correcte MAIS pas avec `Y extends X&_`.

— Globalement j'ai passé du temps un peu partout à choisir les structures qui me conviennent le mieux, et c'est notamment le cas pour la gestion des méthodes et champs (avec les héritages). Comme dit précédemment env_typage.methodes est une Hashtbl qui associe à un identifiant, une `methtab` regroupant toutes ces méthodes. Une `methtab` est aussi une Hashtbl, qui associe à un nom de méthode, toutes les informations intéressantes. Pour le coup on retient vraiment tous les méthodes, y compris celles héritées (en ayant substitués les types) et c'est infiniment plus pratique.

Ce que je ne fais pas pour les relations d'extends par exemple. Pour savoir si `A<U> extends B<K>` je remonte l'arbre d'héritage de `A` à la recherche de `B` à chaque fois. Je ne conserve pas la liste "extends généralisée", ce que j'aurai peut-être dû, mais je l'ai réalisé après avoir écrit les fonctions `verifie_extends`, implements, sous-type etc. Il faudrait faire des modifications un peu partout, et je ne suis pas suffisamment convaincu de la nécessité de changer.

— Les String, Object et System.out.print sont un peu rajoutés au cas par cas là où il le faut. J'hésite à faire un type spécial pour les chaînes de caractères, `Jstring` comme j'ai

`Jint, Jbool, Jtypenull`, en revanche je suis à peu près convaincu qu'un `Jvoid` serait agréable (plutôt que des `jtype (desc) option` partout). À l'heure actuelle, je trouve encore des avantages à voir les chaines de caractères pleinement comme des objets avec une classe "String".

— Mon avant-dernière erreur provenait d'une de mes premières fonctions `verifie_sous_type`, et je pense avoir découvert une erreur dans l'énoncé du sujet (version 3). La règle 5 me semble incomplète, C est un sous-type d'une interface I **si** C implémente J et **J étend I**.

— Autre petite erreur mineure : dans les règles pour le typage des expressions, pour les opérations `==` et `!=`, vous dites qu'il faut l'équivalence entre les deux types. Mais si l'un des deux est de type `Jtypenull` c'est quand même correcte, même si tout type n'est pas sous-type de `Jtypenull`.

Le typage est désormais complet et je vous remercie beaucoup pour tous vos tests. Premièrement parce qu'ils aident à dénicher tout un tas d'erreur plus ou moins importantes, mais aussi parce qu'ils sont très très satisfaisants. De voir que tous ces messages d'erreurs différents servirent, ça fait plaisir!!!

D'ailleurs c'est le gros point noir de mon programme, j'aimerais rendre les messages d'erreurs plus propre. Car pour le moment il ne peuvent renvoyer qu'une seule localisation et des fois deux seraient plus agréables. Exemple : `I c; ... c = A;` avec A non sous-type de I. Plutôt que de cibler uniquement l'expression `c = A`, on pourrait aussi dire d'où le type de c provient.

J'en profite pour vous remercier pour vos cours exceptionnels, et pour dire un grand merci à Naëla. À qui je pose énormément de questions en présentiel et qui m'aide aussi bien pour débuter en Java que pour me servir de mon ordinateur.