Notre projet est en Ocaml, le lexer utilise Ocamllex et le parser Menhir. Nous passons 100% des tests à tous les niveaux. Nous nous sommes organisés ainsi : Théotime a fait le lexer, Samuel le parser, Théotime le typage et enfin Samuel la production de code. Ce qui explique pourquoi nous employons le singulier dans nos parties respectives. Même si dans les faits nous avons réfléchis à deux aux différents problèmes et soucis rencontrés. C'est surtout pour certains aspects purement technique du typage, que j'ai (Théotime) cherché tête baissée.
Nous avons jugé inutile de faire des commentaires sur le lexer et le parser.

0.1 Le typage :

La partie qui suit date de décembre.
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 : en décembre, quand j'ai fini pour la première fois le typage, ne sachant pas à l'avance exactement ce qui est utile pour la production de code, la fonction de typage ne produisait rien. Elle renvoyait une erreur si justifiée, sinon elle disait juste que le typage s'est bien passé. Je préférais ne pas m'encombrer avec ça, au vu du nombre de modifications que j'ai dû 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). Je parle de l'arbre de sortie à la fin de mes commentaires sur le typage.

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 quel env_typage contient quelle information. 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 toujours correcte, même si tout type n'est pas sous-type de `Jtypenull`.

Commentaire final datant de décembre :
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 !!! (avec `make tests`)
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`, nous pourrions aussi dire d'où le type de c provient.

Commentaire final concernant le typage pour le rendu final :

C'était prévisible, je n'ai pas eu le temps de retoucher au typage pour l'embellir (avec le type Jvoid au lieu d'utiliser des jtype option ou en rajoutant des messages d'erreurs plus précis). En revanche désormais le typeur renvoie un arbre de sortie très pratique pour la production de code (la "compilation"). Nous avons fait le point de manière à avoir un maximum d'informations tout en ne gardant rien d'inutile.

- Pour les instructions, on garde le type apparent uniquement pour les accès de champs (ou plus précisément on le fait apparaître puisqu'il n'y était pas). De plus on se débarrasse des paramètres de types et si x est un objet de type T où T est un paramètre de type, on retrouve quelle est la première vraie classe mère de T. Par exemple pour `T extends X, X extends C<Y>, Y extends I`, on remplace X par C et de même on considère tout objet de type Y comme un objet de la classe `Object`.

- Pour les instructions, on sépare le "+" d'addition d'entiers du "+" de concaténation et on a une nouvelle opération unaire signifiant qu'il faut convertir un entier en chaîne de caractères. On rajoute également des "this." dès qu'ils sont sous entendus pour éviter des ambiguïtés. La méthode `equals` de la classe `String` est désormais traitée à part, tout comme `System.out.print` et `println`.

- Les corps des méthodes sont désormais regroupés dans une grande table `tbl_meth`. Excepté pour la méthode main. On accède au corps d'une méthode avec une clé : (*nom de la classe où se trouvait la définition de cette méthode*, *nom de la méthode*). Pour une classe on n'a plus qu'à se souvenir de la liste des clés des méthodes qu'elle peut utiliser, tout comme on garde la liste des noms de ses champs. En revanche les constructeurs restent avec les classes (cf la fin de `ast_typing.ml`).

- On renvoie également la `node_obj` qui est la racine l'arbre des héritages des classes, de manière à garder tout l'arbre (elle nous sert pour la "pré-compilation").

Comme mentionné, nous avons rajouté une étape de pré-compilation / pré-production de code. La seule chose qu'il nous reste à faire après le typage, c'est calculer les offsets pour les champs et les méthodes.

- Pour les champs : lorsqu'on veut accéder à un champ, on connaît sa vraie classe (cf premier tiret du paragraphe précédent). Ainsi on a fait une table d'offset qui à chaque classe et chaque nom de champ, associe l'offset du champ. Pour ce faire on doit simplement suivre l'arbre des héritages (d'où l'utilisé de `node_obj`), de manière à garder les mêmes offsets pour les champs déjà présents dans les sur-classes.

- Pour les méthodes : nous n'avons pas opté pour la même stratégie. En effet même sans relation d'héritage, deux classes peuvent partagées une méthode (si elles implémentent des interfaces en relation), qui doit avoir un unique offset. Ainsi nous avons choisi d'associer à chaque nom de méthodes un offset. Et j'ai (Théotime) cherché une attribution des offsets de manière à minimiser les blancs. Par exemple : si on a une classe A avec une méthode a, idem B avec b et enfin C avec$a$ et b. Peu importe si on attribue l'offset 8 à a et le 16 à b, l'un des descripteurs de classe de A ou de B devra laisser une ligne vide. En revanche : si la classe A contient désormais aussi une méthode d, je les range ainsi : dans le descripteur de A, d en 8 et a en 16, dans le descripteur de B, b en 8 et enfin dans le descripteur de C, b en 8 et a en 16. b et d ont le même offset mais ça ne pose pas de problème puisqu'ils ne se trouvent jamais dans une même classe.
  Pour calculer ces offsets j'ai fait un coloriage de graphe avec une heuristique sortie du chapeau que je détaille dans `pre_compiler.ml`.

0.2 La production de code / compilation :

Le code produit est toujours de la forme:

TEXT	
main	bloc Main
new	constructeur par défaut
classe.new	corps des constructeurs
classe.ident	corps des méthodes
String.equals	méthode String.equals hard-codée
Convert	procédure pour convertir les entiers en string
Printf	alignement de la pile avant appel à "printf"
DATA	
classe	descripteur de classe
Print	formats donnés à printf
string	chaînes de caractère primitives

Problèmes rencontrés et soution apportées:

- Différencier concaténation et addition : communiqué par le typage
- Garantir unicité des étiquettes : tirer avantage des restrictions sur les ident (pas de caractère '.', pas de mots-clés, ...) et identificateur (.int) si besoin (.0 pour le code permanent)
- Libérer la pile des variables locales : place occupée sur la pile passée en argument et non pas globale
- Respecter les extends et implements pour la position des méthodes dans les descripteurs de classe : coloriage dans pre$_c$*ompilerConversiond'entierenstring* : *essaisavecsprintfpuisitoaavantdel*
  cod

- Concaténation : pour calculer la place à allouer sur le tas, on retient la taille d'une string son adresse-8
- Division : le signe de rax doit passer sur rdx (merci Florian et Mandelbrot!)
- Booléen true : notq inverse tous les bits donc on représente true par -1 au lieu de 1
- Différencier l'accés aux méthodes/champs : communiqué par le typage
- System.out.print et String.equals : transformées en expressions spéciales
- Caractères sur un octet et non huit : en manipulant huits octets à la fois on risque de prendre en compte des octets hors de la mémoire allouée pour la string
- Héritage des constructeurs : le comportement est assez obscur en java, on choisit ici d'appeler le parent ssi il n'a pas de paramètres

  Spécificités:

  - Les chaînes de caractères qui apparaissent plusieurs fois dans l'entrée n'ont qu'une occurence dans .data
  - On aligne la pile pour printf mais pas pour malloc et strcat
  - Les conventions registres ne sont pas rigoureusement respectées
  - La première adresse du descripteur pointe vers le contructeur
  - On utilise strcat pour la concaténaton de chaînes de caractères