Rapport global du projet

Livrable 1: Analyse lexicale

Ce livrable a pour but de fournir un code documenté et testé dont la fonction lex et l'objet principal. Cette fonction prend en paramètre un fichier contenant la définition des lexèmes et un fichier source de code assembleur Python. Elle renvoie la liste de lexème contenue dans le fichier source dans l'ordre de lecture.

Ce que nous avons fait durant la période du 07/09/2022 au 21/09/2022 :

Nous avons tout d'abord implémenté les prototypes des fonctions que nous allons utiliser dans notre programme.

Une fois ceci fait, nous avons fait chacun une partie des fonctions et structures nécessaires : Antoine s'est occupé de la structure et des fonctions liées aux chargroups.

Théo s'est chargé de la fonction lisant un fichier d'expression régulières et retournant une liste de définition des lexèmes.

Enfin, Benjamin s'est occupé de faire matcher une chaîne de caractères avec une liste de chargroup et de réaliser la fonction de lecture du code assembleur pour retourner la liste des lexèmes présents.

Où nous en sommes :

Le code en tant que tel est fonctionnel. Nous avons testé les tests lexer du dossier integration à la main pour vérifier le bon fonctionnement global.

Néanmoins, il nous reste encore quelques tâches pour finaliser totalement ce premier livrable. Pour cela, nous sommes actuellement en train de peaufiner et optimiser nos fonctions, certains tests ne sont pas encore fonctionnels (notamment les tests de lexer qui sont en cours de rédaction/modification pour montrer le bon fonctionnement de notre lexer et que l'on puisse les exécuter via **execute_tests.py**). Nous complétons également des commentaires afin de rendre le code plus lisible.

Gestion de projet :

Voici l'organisation de notre gestion de projet ainsi que des commentaires sur chaque partie du code.

• T.1: "Parsing de code assembleur python" (resp. Théo Lafond) : Découpage d'expression régulière tiré d'un fichier contenant une liste de lexèmes.

• T.1.0 “Architecture du code” (resp. Théo Lafond): attribuée 07/09/22 date de fin prévue: 09/09/22 terminée: 09/09/22.

• T.1.1 “module de groupe de char” (resp. Antoine Bach): attribuée: 09/09/22 date de fin prévue: 14/09/22 terminée: 12/09/22 temps consacrée: tout le week-end.
Créer le module chargroup afin de pouvoir lire et interpréter les expressions régulières valide en connaissant la taille

• T.1.2 “module parsing des chargroups” (resp. Antoine Bach) attribuée: 14/09/22 date de fin prévue: 18/09/22 terminée: 16/09/22.
Amélioration du chargroup de base pour qu’il parse les expressions régulières et qu’il retourne une liste de chargroups si l’expression régulière est valide.

• T.1.3 : “parsing du document de définition des lexèmes” (resp. Théo Lafond) attribuée: 09/09/22 date de fin prévue: 15/09/22 terminée: 12/09/22 temps consacrée: un jour et demi.
Créer une fonction qui va lire un fichier contenant les définitions des lexèmes et qui retourne une liste contenant des deflex. Un deflex est une structure composée du type (nom associé à une expression régulière d’un lexème) du lexème et d’une liste de chargroup.

• T.1.4 “module de recherche d’un lexème parmi un liste de définition de lexème” (resp. Benjamin Marty) attribuée: 09/09/22 date de fin prévue: 14/09/22 terminée: 12/09/22.
Création de la fonction re_match_global faisant appel à la fonction re_match et re_match_once agissant respectivement sur une liste de chargroup et un chargroup. Elle prend en paramètre une liste de définition de chargroup et une chaine de caractère et retournant le type de match ou une erreur en fonction de re_match. Les fonctions re_match et re_match_once permettent de naviguer dans la liste de chargroup et de valider ou non une chaine de caractère avec un chargroup.

• T.1.5 “module parsing du fichier assembleur” (resp. Benjamin Marty) : attribuée: 14/09/22 date de fin prévue: 18/09/22 terminée: 18/09/22.
Création de la fonction lex à partir de la fonction doclex et de la fonction re_match_global lisant le fichier source.txt et retournant la liste des lexèmes du fichier source dans l’ordre.

• T.1.6 “test parsing des chargroup” (resp. Antoine Bach) attribuée: 16/09/22 date de fin prévue: 18/09/22 terminée: 18/09/22.
Test d’intégration du chargroup parse avec rajout des règles que l’on a décidé de valider ou non.

• T.1.7 “lexer et fichier test des lexèmes” (resp. Antoine Bach) attribuée: 16/09/22 date de fin prévue: 18/09/22 terminée: 18/09/22.
création du main de lecture du fichier contenant les regexp et lecture du code assembleur à vérifier faisant appel à lexem.c. Premier test à la main du fonctionnement de lexer.

• T.1.8 “test du parsing du document de définition des lexèmes" (resp. Théo Lafond) attribuée: 15/09/22 date de fin prévue: 20/09/22 terminée: 20/09/22.
Test réalisé et réussi. Un fichier source doclex_testing.c a été créé pour pouvoir print le parsing (print qui n’est pas effectué dans la fonction de base. Ce fichier une fois utilisé pour le test est déplacé dans old.

• T.1.9 “fichier de définition des lexèmes" (resp. Benjamin Marty) attribuée: 09/09/22 date de fin prévue: 14/09/22 terminée: 14/09/22.
Fichier contenant d’une part les types correspondant aux expressions régulières et d’autre part les expressions régulières elles-mêmes. Première version contenant un petit ensemble d’expressions régulières récupérées dans le sujet. Version plus complète (push le 20/09/2022).

Choix techniques :

Nous avons décidé (pour ce livrable 1) de limiter le nombre de caractère ASCII à 128 car nous avons souhaité ne pas choisir quel type d’ASCII étendu opter. De plus, ce choix peut être repris assez aisément s’il faut en transformant tous les char en unsigned char.
Nous avons décidé de faire le parsing des regexp au niveau des chargroup.

Tout d’abord, pour un souci de clarté. Il est logique que, comme l’on a à parcourir quoi qu’il arrive les regexp dans chargroup.c, de faire le parsing directement dedans. Il a fallu donc décider de ce qu’on décide d’être valide ou non. Pour voir le détail de ce qu’on autorise ou pas, je conseille de regarder le test 06_weird_KO_or_OK.test dans 01_test_chargroup_parse où j’ai essayé de récapituler les principaux choix (de ce que l’on considère valide ou non) qui ne sont pas forcément logique de prime abord mais que nous avons choisi par souci algorithmique et/ou par choix arbitraire de ce qui nous semblait valable ou non (e.g le caractère impaire de la négation ou l’interdiction ‘]’ sans backslash).

Pour vous montrer l’évolution de nos structures et de notre code global, nous vous avons mis à disposition dans le dossier old les versions initiales de tests/codes sources fonctionnels mais qui ont été changés par le changement de structure que nous avons opéré (i.e le parsing est passé de doclex à chargroup, changement des règles de regexp valide dans chargroup etc).

Livrable 2 : Analyse syntaxique et début de la création des objets python

Notre but dans ce livrable est de d’abord corriger les retours qui nous ont étés faits sur le livrable 1, notamment concernant les tests d’intégrations et unitaires. Le deuxième objectif est d’implémenter l’analyse syntaxique de la source. Et enfin de créer les objets python représentant le header du code pys source.

Ce que nous avons fait durant la période du 21/09/2022 au 05/10/2022 :

Nous avons d'abord réalisé l'analyse syntaxique puis nous avons créé les objets python représentant le header du code pys source. Les deux modules sont fonctionnels (tests effectués au cours du développement). Nous avons donc pu ensuite travailler sur les tests qui nous manquaient dans le livrable 1.

Gestion de projet :

Voici l'organisation de notre gestion de projet ainsi que des commentaires sur chaque partie du code.

• T.2: "Livrable 2" (resp. Antoine Bach) : Livrable 2

• T.2.0: "Architecture du code et du travail sur le livrable 2" (resp. Antoine Bach) attribuée: 21/09/22 date de fin prévue: 24/09/22 terminée: 24/09/22.

• T.2.1.1: "Parseur d'analyse syntaxique d'une liste de lexem" (resp. Antoine Bach) attribuée: 24/09/22 date de fin prévue: 28/09/22 terminée: 25/09/22.

• T.2.1.2: "Test de Parseur d'analyse syntaxique d'une liste de lexem" (resp. Antoine Bach) attribuée: 24/09/22 date de fin prévue: 28/09/22 terminée: 28/09/22

• T.2.2.1.1: "Fonctions d'interfaces basiques avec les objets pythons" (resp. Benjamin Marty) attribuée: 24/09/22 date de fin prévue: 02/10/22 terminée: 02/10/22.

• T.2.2.1.2: "Fonctions lexem to pyobj et list lexem to pyobj" (resp. Théo Lafond) attribuée: 24/09/22 date de fin prévue: 02/10/22 terminée: 02/10/22.

• T.2.2.2: "Test de Fonctions d'interfaces avec les objets pythons" (resp. Benjamin Marty) attribuée: 24/09/22 date de fin prévue: 05/10/22 terminée: 03/10/22.

• T.2.3.1: "Implémentation de la construction des objets pythons dans le parseur" (resp. Benjamin Marty) attribuée: 01/09/22 date de fin prévue: 03/10/22 terminée: 02/10/22.

• T.2.3.2: "Test d'Implémentation de la construction des objets pythons dans le parseur" (resp. Théo Lafond) attribuée: 02/10/22 date de fin prévue: 04/10/22 terminée: 04/10/22.

• T.1.6: "Test intégration regexp et chargroup" (resp. Antoine Bach) attribuée: 02/10/22 date de fin prévue: 04/10/22 terminée: 03/10/22.

• T.1.7: "Test intégration lexer" (resp. Antoine Bach) attribuée: 02/10/22 date de fin prévue: 04/10/22 terminée: 03/10/22.

• T.1.10: “Test unitaires des lexem” (resp. Théo Lafond) attribuée: 02/09/22 date de fin prévue: 04/10/22 terminée: 03/10/22.

• T.1.11: “Test intégration doclex” (resp. Théo Lafond) attribuée: 02/09/22 date de fin prévue: 04/10/22 terminée: 03/10/22.

Choix techniques :

Pour l’analyse grammaticale, nous avons essayé de rassembler au maximum les non-terminaux similaires au niveau structurel pour utiliser des fonctions communes à ces derniers. Cela nous a permis de réduire drastiquement la longueur du code et également rend le code plus lisible et compréhensible. Nous avons décidé que les blancs présents dans les règles grammaticales sont obligatoires et avons ainsi, pour cet effet, créé la fonction blank_or_not dans lexem.c qui permet de vérifier s’il y a un blanc. Nous avons ensuite testé le parser via un jeu de test dans tests/integration pour s’assurer du bon fonctionnement de celui-ci. Ces tests avaient été rédigé en amont pour certains car il nous semblait important de voir les endroits où on aurait des difficultés dès le début. Cela nous a permis de rédiger correctement notre code pour éviter les erreurs qui auraient pues passer inaperçu sans ce jeu de test initial.

Nous avons choisi de remplir le py_codeblock au fur et à mesure de l’analyse syntaxique. Cela permet de parcourir le code qu’une seule fois et puis surtout, la structure de conditions de l’analyse syntaxique est la même que celle dont on aurait besoin pour le remplissage le py_codeblock.

Livrable 3 : Bytecode, lnotab et sérialisation

Dans ce livrable, nous allons générer le bytecode python sans prendre en compte la présence des fonctions dans un script python. Pour ce faire, nous devons tout d’abord rajouter lnotab, un objet python sous forme de string contenant les numéros de lignes ainsi que le nombre d’octets séparant deux lignes contiguës dans le .pys. Enfin, nous allons faire la sérialisation du py_codeblock pour générer le binaire python acceptable par la machine virtuelle python.

Ce que nous avons fait durant la période du 05/10/2022 au 19/10/2022 :

Tout d'abord nous avons ajouté des tests et modifié quelques fonctions des parties précédentes.

Ensuite nous avons travaillé pour faire et finir les tâches du livrable 3. Pour cela, nous avons commencé par modifier la lecture du fichier de définition des lexems pour y ajouter les opcodes et également ajouter les obcodes aux structures de définitions de lexems (deflex) et lexem. Ensuite nous avons pu faire le module pyas permettant de générer le bytecode et le lnotab. Ensuite nous avons fait un fonction de sérialisation qui dans le module py_codeblock qui met dans un fichier l'entièreté d'un py_codeblock. Nous avons pu tester manuellement l'ensemble des réalisations en effectuant des compilations avec le –easy. Ensuite nous avons pu implémenter les labels pour pouvoir compiler sans le –easy.
Et enfin nous avons créé un code python permettant de tester l'ensemble de notre code. Celui-ci fonctionne en compilant un grand nombre de fichiers .py ne contenant pas de fonctions qui ensuite les désassemble en .pys puis exécute notre code sur les .pys. Et enfin, compare les binaires d'origines et les binaires créés par la compilation de notre code.
Tout cela est fonctionnel et testé, il reste tout de même une erreur par rapport à l'internisation des chaînes qui contiennent des caractères spéciaux qui est particulière en python. Nous réglerons cela dans le livrable 4.

Gestion de projet :

• T.3.1 "Ajout de l'opcode dans doclex et deflex et lexem et test" (resp. Lafond Théo) attribuée: 05/10/22 date de fin prévue: 08/10/22 terminée: 07/10/22.
Maintenant dans le fichier de définition des lexems est inscrit l'opcode associé aux instructions. Et celui-ci est ajouté aux structures de définition des lexems (deflex) et la structure lexem afin de pouvoir facilement accéder à l'opcode lors de la génération du bytecode.

• T.3.2 "Module pyasm : lnotab & bytecode" (resp. Bach Antoine) attribuée: 06/10/22 date de fin prévue: 12/10/22 terminée: 12/10/22.
Mis en place par double lecture de lnotab et bytecode:
-première lecture calcul des tailles des string lnotab et bytecode
-deuxième lecture remplissage des strings lnotab et bytecode correctement allloués

• T.3.3 "Sérialisation de l'objet python" (resp. Marty Benjamin) attribuée: 10/10/22 date de fin prévue: 15/10/22 terminée: 13/10/22.
Sérialisation avec notamment la prise en compte des variables internées.

• T.3.4.1 "Traitement des labels dans le pyasm" (resp. Lafond Théo) attribuée: 12/10/22 date de fin prévue: 18/10/22 terminée: 18/10/22.

• T.3.5 “Test global du code” (resp. Lafond Théo) attribuée: 16/10/22 date de fin prévue: 18/10/22 terminée: 18/10/22.
Nous avons créé un code python permettant de tester l’ensemble de notre code. Celui-ci fonctionne en compilant un grand nombre de fichiers .py ne contenant pas de fonctions qui ensuite les désassemble en .pys puis exécute notre code sur les .pys. Et enfin, compare les binaires d’origines et les binaires créés par la compilation de notre code.
Tout cela est fonctionnel et testé, il reste tout de même une erreur par rapport à l’internisation des chaînes qui contiennent des caractères spéciaux qui est particulière en python. Nous réglerons cela dans le livrable 4.

Pour l'exécuter, se placer dans /tests/global et exécuter ‘python test_global.py’

• T.2.1.1 “Début start code, names en option, changement parsing code” (resp. Bach Antoine) attribuée: 05/10/22 date de fin prévue: 18/10/22 terminée: 17/10/22.
Récupération de l’adresse du début du code de l’assembleur via pointeur. Changement de la grammaire du parser pour mettre names en option et non en obligatoire. Enfin, changement de la grammaire du parsing du code assembleur pour vérifier que l’instruction de fin est un return value, les instructions sont forcément dans une ligne ou un label et les labels sont forcément dans une ligne.

• T.2.2.1.2 “Tuple in pyobj” (resp. Lafond Théo) terminée: 06/10/22.

• T.1.3 “Changement du lexer et optimisation” (resp. Lafond Théo) attribuée: 01/10/22 date de fin prévue: 18/10/22 terminée: 08/10/22.
Changement du lexer en passant notamment le fgetc en lecture ligne par ligne afin d’avoir un code plus lisible et plus optimisé. Pour la lecture ligne par ligne nous avons choisis d’utiliser getline() et pas fgets pour qu’il soit possible de lire une ligne de taille maximale indéterminée.

• T.2.4 “Changement de py_codeblock vers py_cb” (resp. Marty Benjamin) attribuée: 18/10/22 date de fin prévue: 18/10/22 terminée: 18/10/22.

Choix techniques :

Pour simplifier la manipulation des données au sein de notre code, nous avons choisis de rajouter les opcodes directement dans la définition de nos lexèmes. Nous réalisons cette opération lors de la lecture du fichier definitionLexeme.txt. Ainsi, lorsque nous voulons réaliser l’objet python string contenant les éléments du bytecode, nous n’avons plus qu’a accédez à l’éléments opcode de notre lexème.

Livrable 4 : Fonctions et bug

Maintenant que la sérialisation fonctionne pour des codes sans fonctions, nous pouvons maintenant finir en les prenant en compte. Et ensuite, le code complet sera fini, reste plus qu'à tester sur beaucoup de codes python pour détecter d'éventuels problèmes.

Ce que nous avons fait durant la période du 19/10/2022 au 10/11/2022 :

Nous avons d'abord, assez rapidement introduit la récursivité pour que les fonctions soient compilées. Nous avons également ajouté le traitement des tuples et des complexes. Puis pendant les vacances nous avons testé notre code en comparant le binaire original avec celui généré par notre code. Nous avons découvert plusieurs problèmes auxquels on ne s'attendait pas et nous les avons résolus, ils sont décrits plus bas dans les tâches. Et enfin nous avons mis notre code au propre et généré une documentation doxygène.

Gestion de projet :

• T.4.1 "Traitement des fonctions par récursivité" (resp. Tout le monde) attribuée: 19/10/22 date de fin prévue: 26/10/22 terminée: 30/10/22.
• T.4.1.1 "Tuples" (resp. Théo Lafond) attribuée: 19/10/22 date de fin prévue: 06/11/22 terminée: 03/11/22.
• T.4.1.2 "Complexes" (resp. Antoine Bach) attribuée: 19/10/22 date de fin prévue: 06/11/22 terminée: 06/11/22.
• T.4.2 "Test globaux avec beaucoup de codes pythons" (resp. Théo Lafond) attribuée: 31/10/22 date de fin prévue: 03/11/22 terminée: 02/11/22.

Puis la correction de bug découverts :
• T.4.3 "Deux premiers octets de lnotab" (resp. Antoine Bach) attribuée: 02/11/22 date de fin prévue: 08/11/22 terminée: 06/11/22.

Nous avons d'abord remarqué lors de la mise en place de la prise en compte des fonctions python que le lnotab des fonctions avait toujours les deux premiers octets du début. En effectuant des tests, nous avons remarqué que ce n'était pas le cas pour les fonctions lambda en python. Et que en général c'était le cas pour toutes les fonctions qui avaient leurs premières instructions à la même ligne que leur déclaration.

• T.4.5 "Interned ou string ref" (resp. Benjamin Marty) attribuée: 02/11/22 date de fin prévue: 08/11/22 terminée: 05/11/22.

Nous avons remarqué que l'internalisation des string n'étais pas aussi simple que de simplement mettre t au lieux de s si la string était dans les interned. Mais que si elle est dans interned et qu'elle a déjà était internée dans la sérialisation, alors on fait une string ref (R).
Pour mettre cela en place nous avons créé la structure already_interned qui stock le pyobj d'une string internée et un booléen qui permet d'enregistrer si on l'a déjà internée ou pas.

• T.4.6 “Return value pas à la fin” (resp. Antoine Bach) attribuée: 02/11/22 date de fin prévue: 08/11/22 terminée: 06/11/22.

Dans une fonction on peut trouver des return value autre part qu’à la fin des instructions. Nous avons donc dû changer la grammaire du parseur.

• T.4.7 “Echappement \n et \t dans les strings de la source” (resp. Théo Lafond) attribuée: 02/11/22 date de fin prévue: 08/11/22 terminée: 04/11/22.

Problème réglé dans lex.

• T.4.8 “Documentation et Doxygene” (resp. Théo Lafond) attribuée: 06/11/22 date de fin prévue: 09/11/22 terminée: 09/11/22.

Rapport Global

Bilan global :

L’objectif de ce projet était de re-générer un fichier binaire à partir d’un fichier assembleur lui-même généré à partir d’un fichier binaire python grâce à pyc-objdump. Voici donc les différentes étapes que nous avons accomplies avant de pouvoir avoir notre propre fichier binaire.

Livrable 1 :

Tout d’abord, nous devions pouvoir classer chaque lexème du fichier assembleur selon leur nature afin de générer une liste de lexèmes de notre fichier assembleur. Afin d’accomplir cette tâche, nous avons construit, grâce au sujet, un fichier txt appelé definitionLexeme.txt contenant le nom du lexème ainsi que sa représentation au format d’expression régulière. Afin de pouvoir manipuler plus aisément les expressions régulières, nous avons créé le type chargroup_t. Ce type contient un tableau de 128 char initialisés à 0 ainsi qu’un type énuméré correspondant à l’opérateur associé. Si nous devons créer un chargroup contenant la valeur ‘t’, alors nous “allumons” la case du tableau correspondante en faisant : tableau[‘t’] = 1. Cependant, sachant qu’une expression régulière pouvait regrouper plusieurs chargroup, nous avons décidé de créer un module complet pour chargroup. Ainsi, à partir d’une expression régulière, nous pouvons maintenant générer une liste de tableaux correspondant à cette dernière. C’est pourquoi, en parcourant le fichier definitionLexeme.txt, nous avons pu créer une liste contenant toutes les définitions des lexèmes connues. Si certaines venaient à manquer, il suffisait de rajouter son nom et son expression régulière dans le fichier txt. Après ces étapes, la génération de la liste des lèxemes du code assembleur se fut sans grande difficulté.

Livrable 2 :

En deuxième temps, nous devions faire une analyse syntaxique de notre code assembleur. La raison pour laquelle nous avons généré une liste de lexèmes est donc évidente. Il est en effet plus facile de parcourir une liste qu’un fichier. Pour accomplir cette tâche, nous avons suivi une règle grammaticale en O(1) (L’élément n dépend uniquement de l’élément n-1) dont l’inspiration nous vient directement du sujet. Nous précisons cependant que certains changements ont étaient nécessaires dû à certaines imprécisions présentes dans le sujet. Parallèlement à ceci, nous avons créé le type pyobj ainsi que le type

py_codeblock. Le type pyobj correspond à un objet python. C'est d'ailleurs ici que nous avons compris le problème principal de complexité de python. En effet, faire en sorte que n'importe quelle variable, valeur etc sont stockées sous le même type implique différentes conséquences. Notamment la place prise par une variable. Le pyobj étant une union de plusieurs types, la place allouée pour un simple caractère ou bien une string reste la même, à savoir la taille max retrouvée dans l'union. Nous précisons néanmoins que cela n'apporte pas que des aspects négatifs, la simplicité de python découle également de ce choix là. Le type py_codeblock est quant à lui représentatif d'un objet de type code. Il est constitué de multiple pyobj représentant chacun un élément différent tel que le nom du fichier, les différentes variables à internées ou encore un timestamp. Nous pouvons donc, avec ces différents types, sérialiser nos codes.

Nous avons commencé par générer nos premiers fichiers assembleur avec l'option –easy, c'est pourquoi nos premières sérialisation furent très facile. En effet, connaissant l'ordre d'apparition des éléments d'un py_codeblock ainsi que la manière dont ils sont écrits, nous n'avions plus qu'à créer une fonction réalisant ce travail. Il nous a simplement fallu modifier légèrement la structure d'un lexème ainsi que notre fichier definitionLexeme.txt afin d'avoir accès à l'opcode directement. Par ailleurs, l'aide apportée par le sujet nous soulagea d'heures de travail acharné devant des octets à comprendre l'ordre d'affichage des différents éléments d'un py_codeblock. Ensuite, nous avons créé une structure pour remplacer nous même les valeurs des labels par leur valeur respective. Nous avons donc dû parcourir toute la liste de lexèmes du fichier assembleur pour calculer les valeurs. Durant ce parcours, nous avons également compléter notre objet python avec l'objet firstlinelo, un tableau contenant le nombre d'octet variant de ligne en ligne. Ce dernier n'étant pas créé lors de la vérification lexicale, nous avons estimé judicieux de le faire en même temps que la gestion des labels. A la fin de cette section ci, nous avions une sérialisation correcte mais pas parfaite. Nous pouvions exécuter nos binaires mais nous n'avions pas une exacte similitude entre le fichier initial et notre fichier. Pour comparer ceci, nous avons implémenté un code python effectuant la ressemblance entre nos deux fichiers (en ignorant la différence du au timestamp, cela va de soi). En plus de cela, il nous manquait encore la gestion des fonctions.

Les fonctions ne nous posèrent pas de réels problèmes. Nous avions vite compris que, lorsque nous détectons l'instruction code_start, alors nous allons devoir faire un appel récursif à la fonction générant un py_codeblock jusqu'à l'instruction code_end. Quelques différences se faisaient dans la sérialisation. C'est pourquoi durant cette période, la correction de léger décalage lors de la sérialisation nous a pris la majeure partie de notre temps. Notamment la gestion de l'internement des string ou non, pourquoi certaines l'étaient et d'autres non ainsi que la prise en compte des tuples et des complexes.

Ainsi, après près de 2 mois de travail, nous avons réussi à obtenir pour tous nos tests (83 codes pythons dont les caractéristiques sont différentes) un fichier binaire correspondant exactement au fichier binaire initial (moyennant le timestamp).

Bilan sur la gestion de projet :

Chaque soir même ou lendemain de tutorat, nous nous rassemblons pour discuter de l'architecture du code, écrire les prototypes et se répartir les tâches.
Tout le monde était autant investi et motivé.
Antoine : 10/30
Benjamin : 10/30
Théo : 10/30

Ce que nous pourrions rajouter :

- Les caractères spéciaux (>128)
- Les int 64 (si pyc-objdumb les tronquait pas)
- Et les échappements de “ dans les strings

Cette dernière tâche serait la plus pertinente à implémenter mais demanderait de faire un bon nombre de changements. Nous avons imaginé deux possibilités pour l’implémenter.
La première serait de rajouter un nouvel opérateur dans les regexps qui serait l’accolade. L’accolade permettrait de considérer une regexp comme un caractère dans une regexp.
Exemple : a[b{cd}] génère ab et acd.
Cela pourrait permettre de réécrire la regexp de la string : “^[\n{^\\”}]*”
Pour mettre ce nouvel opérateur il faudrait redéfinir le type chargroup en y rajoutant une liste de regexp.

La deuxième solution pour échapper les \” serait de retirer le lexeme string et de le remplacer par les lexems guimet et antislash et le calcul des string se ferait dans le parseur. A la même manière que les tuples.

La deuxième option serait plus simple à mettre en œuvre mais moins polyvalente et élégante que la première.

Annexe: nouvelle grammaire en EBNF du langage assembleur Python

<pys> : := <eol> <prologue> <code>

<prologue> : := <set-directives> [<interned-strings>] <constants> [<names> <varnames> <freevars> <cellvars>]

<set-directives> : := <set-version-pyvm> <set-flags> <set-filename> <set-name> [<set-source-size>] <set-stack-size> <set-arg-count> [<set-kwonly-arg-count>] [<set-posonly-arg-count>]

<set-version-pyvm> : := {‘dir::set’} {‘blank’} {‘version_pyvm’} {‘blank’} {‘integer::dec’} <eol>

<set-flags> : := {‘dir::set’} {‘blank’} {‘flags’} {‘blank’} {‘integer::hex’} <eol>

<set-filename> : := {‘dir::set’} {‘blank’} {‘filename’} {‘blank’} {‘string’} <eol>

<set-name> : := {‘dir::set’} {‘blank’} {‘name’} {‘blank’} {‘string’} <eol>

<set-source-size> : := {‘dir::set’} {‘blank’} {‘source_size’} {‘blank’} {‘integer::dec’} <eol>

<set-stack-size> : := {‘dir::set’} {‘blank’} {‘stack_size’} {‘blank’} {‘integer::dec’} <eol>

<set-arg-count> : := {‘dir::set’} {‘blank’} {‘arg_count’} {‘blank’} {‘integer::dec’} <eol>

<set-kwonly-arg-count> : := {‘dir::set’} {‘blank’} {‘kwonly_arg_count’} {‘blank’} {‘integer::dec’} <eol>

<set-posonly-arg-count> : := {‘dir::set’} {‘blank’} {‘posonly_arg_count’} {‘blank’} {‘integer::dec’} <eol>

<interned-strings> : := {‘dir::>nterned’} <eol> ({‘string’} <eol>)*

<constants> : := {‘dir::consts’} <eol> (<constant> <eol>)*

<constant> : := <function> | <tuple> | <list> | <complexe> | {‘integer’} | {‘float’} | {‘string’} | {‘pycst’}

<function> : := {‘dir::code_start’} <eol>. <pys> {‘dir::code_end’}

<tuple> : := {‘paren::left’} ({’blank’} <constant>)* [{’blank’}] {‘paren::right’}

<list> : := {‘brack::left’} ({’blank’} <constant>)* [{’blank’}] {‘brack::right’}

<complex> : := {‘complex’} ({‘float’} | {'integer'}) 'j'

<names> : := {‘dir::names’} <eol> ({‘string’} <eol>)*

<varnames> : := {‘dir::varnames’} <eol> ({‘string’} <eol>)*

<freevars> : := {‘dir::freevars’} <eol> ({‘string’} <eol>)*

<cellvars> : := {‘dir::cellvars’} <eol> ({‘string’} <eol>)*

<code> : := {‘dir::text’} <eol> (<assembly-line> <eol>)*

<assembly-line> : := <source-lineno> (<insn> <eol>)+ | <source-lineno> (<label> (<insn> <eol>)+)+

<label> : := {‘symbol’} {‘blank’}* {‘colon’}

<source-lineno> : := {‘dir::line’} {‘blank’} {‘integer::dec’}

<insn> : := {‘insn::0’} | {‘insn::1’} ({‘integer::dec’} | {‘symbol’})

<eol> : := ([{‘blank’}] [{‘comment’}] {‘newline’} [{’blank’}])+