RASCOL Laura | AKIKI Melissa | DUCHADEAU Romain

Compte rendu final

Ce compte rendu permet de mettre en lumière les choix que nous avons fait, la manière de tester les fonctions écrites, l'état d'avancement du projet ainsi que la répartition du travail dans l'équipe pour la finition du livrable 1, du livrable 2, le travail sur le livrable 3 ainsi que celui sur le livrable 4 auquel nous avons beaucoup réfléchi. Mais au vu de l'avancée compliquée pour le livrable 3, nous avons préféré se concentrer sur celui-ci afin de rendre un livrable 3 propre (gestion des labels, libération de la mémoire et exécutable python qui compile).

I Readme

##Projet sei AKIKI-DUCHADEAU-RASCOL
##---

note: toutes les fonctions test "présentables" n'ont pas de numéro. Celles dont le nom finit par 2 (ex: test-objet2.c) sont les fonctions qui nous permettent de debugguer.

##############################EXÉCUTABLES#################################

#./prog/lexer.c
But: le programme lexer qui réalise l'analyse lexicale d'un fichier .pys passé en argument sur la ligne de commande
Tester: Tapper "./bin/lexer config/pyas.conf.txt ./functions/function[i].pys" (dépends de quelle fonction vous désirez tester)

#./prog/regex-read.c
But:le programme qui lit (parse) une expression régulière passée en argument sur la ligne de commande et affiche sa structure
Tester: Tapper "./bin/regexp-read [expression reguliere]"

#./prog/parser.c
But:le programme parser qui réalise l'analyse syntaxique d'un fichier .pys passé en argument sur la ligne de commande
Tester: Tapper "./bin/parser config/pyas.conf.tx ./functions/function[i].pys"

#./prog/pays.c (/*Fonctionnel sauf pour le livrable 4*/)
But: le programme pays qui réalise l'assemblage d'un fichier .pys passé en argument sur la ligne de commande et génère le fichier binaire pays.pyc correspondant
Tester: -Tapper "./bin/bin config/pyas.conf.txt ./functions/function[i].pays" (dépends de quelle fonction vous désirez tester)
-Pour compiler l'executable python ainsi obtenue, tapper "python ./bin/pyas.pyc"

#./prog/regex-match.c (/*Fonctionnel*/)
But: le programme qui match une expression régulière et une chaîne passées en argument sur la ligne de commande et affiche le résultat
Tester: Tapper "./bin/regexp-match [expression regulier] [source a tester]" (ex:[a-c]+ aaab)

##
#config/pyasm.conf
But: fichier regroupant tous les types avec les définitions associées; c'est le dictionnaire complet de lexem

#database.txt.test
But: Dictionnaire de lexem réduit pour les tests

#queue.c
But: Définir toutes les fonctions nécessaires à l'utilisation des queue
Avancement: Tout fonctionne parfaitement
Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-queue.exe -v " ou pour tester avec le type char group " ./bin/unit/test-queue-cg.exe -v ".
Cette commande va tester les fonctions : queue_new, queue_empty, enquête, (queue_to_list), queue_peek, dequeue, queue_length.

#list.c
But: Définir toutes les fonctions nécessaire à l'utilisation des queue
Avancement: Tout fonctionne parfaitement
Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-list.exe -v " qui a été modifié après le bootstrap
ou pour tester list compare Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-list2.exe -v " Fonctionnelle

#char group.c
But: Définir toutes les fonctions nécessaire à l'utilisation des chars group
Avancement: Tout fonctionne parfaitement
Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-char_group.exe -v "

#database.c

But: Lire le config/pyas.conf.txt, et inclure les informations triée dans une queue

Avancement: Fonctionnel

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-database2.exe config/pyas.conf.txt", pour afficher tout ce que fait database.c à partir du fichier complet database.txt.

#regexp.c

But: Comparer une source char* avec une expression régulière sous forme char*

Avancement: 3 tests sur 5 fonctionnel (mais réglages de ces problèmes sur la version adaptée à nos choix, c'est à dire char group)

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-regexp.exe ".

#regexp-cg.c

But: Comparer une source char* a une liste de char group

Avancement: Fonctionnel

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-regex-cg.exe [une expression régulière] [une source a comparer avec]". exemple [a]+xa aaax

Pour tester avec les test d'intégrations : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./tests/integration/execute_tests.py runtest ./bin/unit/test-regexp-cg.exe ./tests/integration/02_test_regexp_match "

#parse.c

But: Prendre un char* en argument et mettre les différentes unités logiques dans une liste de chargroup

Avancement: Fonctionnel

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-re-parse.exe   config/pyas.conf.txt.test ".

le programme va prendre le fichier config/pyas.conf.txt.test et mettre les différents éléments char-group dans une liste avant d'imprimer la liste

#lexer.c

But: Elément final: Lire un fichier assembler .pys et renvoyer une liste de lexem

Avancement: Fonctionnel

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-lexer.exe config/pyas.conf.txt.test ./functions/function1.pys ".

#lexem.c

But: Fonctions utiles pour grammar.c

Avancement: Fonctionnel

Pour tester les fonctions du livrable 2: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-lexem.exe ".

Pour tester lexem_compare : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-lexem.-compare.exe ".

RASCOL Laura | AKIKI Melissa | DUCHADEAU Romain

#grammar.c

But: Fonctions pour chaque non-terminal : Fonctions qui vérifient la syntaxe en renvoyant 1 ou 0
Avancement: fonctionnel d'après les tests "à la main" effectués avec test-grammar2.c (utile pour debugger) mais le test_grammar.c (qui est plus exhaustif) ne fonctionne pas encore totalement correctement.
Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-grammar.exe -v" (test-grammar2.c était utile pour debugguer)

#objet.c

But: Fonction qui remplit la structure python : Construction d'un objet de code python
Avancement: fonctionnel
Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-objet.exe"

#production_bytecode.c

But: Fonction qui remplit le lnotab et le bytecode de la structure python
Avancement: fonctionnel avec function1.pys

Pour tester la fonction opcode_is : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-opcode_is.exe -v" Fonctionnelle.

Pour tester la fonction argument_yes_or_no : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-argument-yes-or-no.exe -v" Fonctionnelle.

Pour tester la fonction find_the_part_instruction : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-find_the_part_instruction.exe -v" Fonctionnelle

Pour tester la fonction length_calculated_for_bytelist : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-length-calculated-for-bytelist.exe " Fonctionnelle

Pour tester la fonction lenght_lnotab : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-lenght-lnotab -v" Fonctionnelle

Pour tester la fonction pyobj_from_lnotab_tab : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/pyobj-from-lnotab-tab" Fonctionnelle

Pour tester la fonction pyobj_from_bytecode_list : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/pyobj-from-bytecode-list" Fonctionnelle

Pour tester les fonctions sur la structure du dictionnaire choisie ici label_adress_t : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-dictionnaire.c" Fonctionnelle

Pour tester la fonction pyasm : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-production-bytecode-avec-livrable-1.exe config/pyas.conf.txt functions/function1.pys" Fonctionnelle

ou Pour tester de manière moins exhaustive la fonction pyasm : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-production_bytecode.exe"

#write_bitecode.c

But: Fonction qui ecrit sur un fichier en binaire l'exécutable de la machine virtuelle python

Avancement: Fonctionnel

Pour tester la fonction pyobj_write : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-write_bitecode.exe config/pyas.conf.txt ./functions/function[i].pys" , un message de 'write to file ok' indiquant que l'écriture sur le fichier a été faite .

-Ensuite pour visualiser le fichier "write_bitecode.pyc" de l'éxecutable écrit en binaire tapper " hexdump -C write_bitecode.pyc"

-Pour compiler l'executable python ainsi obtenue, tapper "python write_bitecode.pyc"

II Répartition des tâches

A) LIVRABLE 1

— T.1 : "Parsing des expressions régulières" : Faire les finitions livrable 1.

-T.1.1 : “Libération de la mémoire dans les exécutables”
-T.1.1.1 : Libération mémoire dans livrable_1_lexer.c (resp AKIKI Melissa). (espéré 1 semaine, réel 3 semaines).

La libération de mémoire dans le lexer a été compliquée à perfectionner, vu que dans cette fonction, on fait appel à d'autres, notamment le re_read et re_match (des grosses fonctions). Nous avons dans un premier temps libérer la mémoire des callocs que nous avons créé au début de la fonction, après avoir libérer parfaitement la mémoire venant de re_parse.c et database.c, nous nous sommes concernés à libérer les listes et queues temporaires dont nous avons eu besoin pendant l’algorithme de la fonction. Tout cela avant de se rendre compte qu’il y avait des fuites de mémoire dans regexp_cg.c. (abordés dans la section suivante). Résultat final: réussi.

-T.1.1.2 : Libération mémoire dans livrable_1_regexp_read.c (resp AKIKI Melissa). (espéré 1 semaine, réel 3 semaines).

Au tout début, la libération de mémoire s'est faite en libérant le calloc fait dans la fonction 'asci'. Ce calloc notamment, se faisait à chaque appel de la fonction re_match, d'où le besoin de le libérer à chaque 'return' (une fois l'appel de la fonction est fini). Juste pour se rendre compte, qu'il y avait encore des fuites de mémoires quand l'appel de rematch se faisait avec un 'list_next(regexpp)' dans un return. En effet, ce qui se passait c'était qu'il y avait deux fois collocs pour le caractère 'c' initialisé au début de 're_match' et on n'avait plus accès à l'ancien pour le 'free'. Résultat final: réussi.

-T.1.1.3 : Libérer mémoire dans livrable_1_parser.c (resp RASCOL Laura et AKIKI Melissa et Romain Duchadeau).

Un gros problème rencontré dans cette fonction est que pour remplir un élément dans une liste il faut lui donner un espace mémoire (et donc l'allouer) et bien évidemment supprimer l'espace mémoire à la fin. Une fonction intermédiaire à due être créée pour faire à la fois le free de l'élément d'avant et l'allocation du nouvel élément.

-T.1.1.4 : Libérer mémoire dans database.c (resp DUCHADEAU Romain). Relativement facile

-T.1.2 : "Correction des tests d'intégrations" (resp DUCHADEAU Romain et AKIKI Melissa)

Notamment pour pouvoir tester le bon fonctionnement de regexp_cg.c et re_parse.c, nous avons testé nos fonctions avec les tests d'intégrations. Et grâce à eux, nous avons pu améliorer nos fonctions et régler pleins de petits problèmes.

-T.1.2.1 : "Correction de regexp_cg.c" (resp AKIKI Melissa) (espéré 2 semaines, réel 3 semaines).

Les fonctions dans regexp_cg.c ont été adaptées pour les cas particuliers, notamment pour les sources vides etc… Encore pour la lecture correcte de '\n' et '\t', cela par contre a été fait en correspondance avec les fonctions de 're_parse.c'.
Sans oublier que re_match avant, comparait mal les '^'. Cela est normalement fixé à présent.
'regexp_cg' est donc adapté à la comparaison des expressions régulières les plus usuelles.

-T.1.2.2 : "Correction de re_parse.c" (resp DUCHADEAU Romain) . Un travail de fond effectué en parallèle du reste. La correction des tests d'intégration à permis de se rendre compte de beaucoup d'erreurs de parsing mais certaines erreurs restent à régler. Une grosse partie des erreurs rémanentes sont dûes à des cas non pris en compte par le parser car jugée grammaticalement fausse.

Un nouveau parser (re-parse-livrable1) à été créé pour coller au modèle fourni. L'ancien parseur aurait totalement pu être réutiliser mais comme il était appelé dans beaucoups de fonctions (des livrables supérieurs notamment) j'ai préféré recréer une fonction pour pouvoir changer le prototype et débugger plus facilement. Si c'était à refaire j'aurais directement choisi le prototype demandé plutôt que la solution la plus naturelle pour moi (qui marche très bien par ailleurs). Nous pourrions très bien se passer de cette fonction additionnelle.

B) LIVRABLE 2

— T.2 "Développement d'un parser" : Produire une analyse syntaxique, afin de voir si les phrases sont correctes, et les stocker dans un objet python

-T.2.1 : Suppression des warnings" (resp Duchadeau Romain).

-T.2.2 : "Création nouveau type pyobj pour bytecode et lnotab" (resp Duchadeau Romain et Rascol Laura).

-T.2.2.1 : "Fonction print de ce type" (resp Duchadeau Romain et Rascol Laura).
-T.2.2.2 : "Allouage de la mémoire" (resp Duchadeau Romain).
-T.2.2.3 : "Libération de la mémoire pour ce type" (resp Duchadeau Romain et Rascol Laura).

-T.2.3 : "ajout du livrable 4 dans la fonction de parsing" (resp Duchadeau Romain). Commencé, pas encore fonctionnel.

-T.2.4 : Libération de la mémoire dans le reste des fonctions du livrable 2"(resp Duchadeau Romain). Franchement pas faciles avec toutes les structures et fonctions imbriquées et la gestion des listes. Énormément de temps passé sur cette partie. Maintenant fonctionnel à priori.

C) LIVRABLE 3/LIVRABLE 4

T.3 "Sérialiser (écrire dans un fichier binaire) l'objet Python"

T.3.1 "Générer le bytecode et lnotab (fonction pyasm(pyobj_t code))" (resp. RASCOL LAURA) : . (espéré 1 semaine, réel 2 semaines).

-T.3.1.1 : 'Faire fonctionner pyasm sans les label' (espéré 1 semaine, 2 semaines) (resp Laura Rascol) Fonction : int pyasm(pyobj_t code, list_t lexem) Le but était de remplir code->py.codeblock->binary.content.bytecode | code->py.codeblock->binary.trailer.firstlineno | code->py.codeblock->binary.trailer.lnotab.

-T.3.1.1.1 : "Ecriture fonction int lexem_compare_also_column_and_line(lexem_t lex1, lexem_t lex2) in lexem.c (+test dans test-lexem-compare.c)" Cette fonction fait pareil que lexem compare mais compare aussi le numéro de ligne et de colonne. (resp Laura Rascol), finalement elle n'a pas été utilisée et la technique pour savoir si le label était appelé ou bien définit était de regarder son numéro de colonne (dans lex->column).

-T.3.1.1.2 : "Ecriture fonction lexem_t lexem_peek_pass_new_line(list_t *lexems)" Fonction qui fait pareil que lexem_peek mais qui passe les newline (comme ca faisait avant poru les comments) (resp Laura Rascol)

-T.3.1.2 : “Ecriture de test-production-bytecode-avec-livrable-1, une fois qu'on a la vraie liste de lexem produite par le livrable 1” (resp Laura Rascol) Cette fonction test permet de tester la fonction pyasm avec la liste produite par le livrable 1.

-T.3.1.3 : “Modifier pyasm pour faire fonctionner les label” (espéré 4h, réel 1 semaine et demi, dont plusieurs, nuits blanches) (resp Laura Rascol)

-T.3.1.3.1 : “Relever les erreurs dans lnotab et dans bytecode” (espéré 1h, réel 2h) (resp Melissa Akiki) Faire une comparaison entre ce qui est produit par write_bytecode et le fichier .pyc produit par le terminal.

-T.3.1.3.2 : “Reverse engineering, comprendre exactement ce qui est attendu” (espéré 1h, réel 8h) (resp Laura Rascol)

Comprendre l’erreur dans lnotab qui venait du fait que je stockais le premier delta line soit 0, et le dernier delta octet, le diaporama du livrable 3 m’a induit en erreur quand à la longueur du lnotab.

Comprendre l’erreur dans bytecode qui venait d’un problème dans le livrable 1 (qui ne captait plus les symbol), mais pas que. En effet au départ je n’avais pas compris, j’allais chercher les lexem des label et je les mettais à l’endroit du label et non à la fin du bytecode. J’ai donc eu du mal à comprendre le sujet. Et le système de reverse engineering m’a permis de bien m’approprier ce qui était attendu. et de mieux comprendre ce qui était attendu au niveau des adresses en fonction des opcodes précédents l’appel aux labels.

Pour ce faire j’ai créé une structure type dictionnaire, label_adress_t qui permet de stocker le nom du label dans name, l’address_first (adresse où il est appelé), et l’address_end (adresse où il est défini). Cette structure est créée dans une queue : en effet c’est le plus simple, car c’est circulaire, et si les label ne sont pas appelé dans le bon ordre, avec une simple boucle while, il sera facile de retrouver le label concerné.

Pour ce qui est de la gestion des sauts relatifs, et du jump forward, si il y a un saut relatif mettre au niveau de l’address_first 0, pour ne pas calculer de delta lors de l’assignation des intger dans bytecode. Si il y a plusieurs appels d’un label à la suite du jump forward, il faut le mettre plusieurs fois dans la queue et y assigner la bonne valeur au niveau de address_first à chaque fois. Lors de l’ajout au bytecode, j’ai créé une condition afin de prendre l’information associée au bon appel au label (puisque c’est possible de l’appeler plusieurs fois).

Comparer le bytecode, le lnotab de plusieurs fonctions . (ce qui a impliqué de créer de nouvelles fonction.py afin de faire un test un peu plus exhaustif).

-T.3.1.3.3 : “Modification de toutes les fonctions liées à lnotab et bytecode” (espéré 1h, réel 4h) (resp Laura Rascol)

-T.3.1.3.4 : “Modification des tests de toutes les fonctions liées à lnotab et bytecode” (espéré 1h, réel 1h) (resp Laura Rascol)

-T.3.1.3.5 : “Test sur la structure adress_label_t dans test-dictionnaire.c” (espéré 1h, réel 1h) (resp Laura Rascol)

-T.3.1.4 : “Libérer la mémoire liée à pyasm” (espéré 4h, réel 3 jours) (resp Laura Rascol)

T.3.2 "Écrire sur un fichier binaire (fonction pyobj_write(FILE *fp, pyobj_t obj))” (resp. AKIKI Melissa) : . (espéré 1 semaine, réel 2 semaines).

-T.3.2.1 :Ecriture de la fonction ‘parsing_type(FILE *fp, pyobj_t obj)’ (resp. AKIKI Melissa) (espéré 1 semaine, réel 2 semaines).

Fonction récursive qui a pour but d'écrire sur le fichier binaire selon le type de l’objet. Notamment, l’adaptation de cette fonction a été faite en correspondance avec les fonctions de remplissage de l’objet python dans le livrable 2. c-a-d, reconnaître comment l'objet python a été remplie, pour pouvoir le sérialiser sur un fichier binaire en tenant compte de la récursivité. En effet, les types ‘Null, none, true, false, int, int64, tuple,string,float,complexe,list et code ont été traités'.Cette fonction sera appelée dans pyob_write.

-T.3.2.2 :Ecriture de la fonction finale ‘pyobj_write(FILE *fp, pyobj_t obj)’ (resp. AKIKI Melissa)
(espéré 1 semaine, réel 2 semaines).

Cette fonction a pour but d'écrire sur un fichier binaire l’objet python dans l’ordre avec lequel la machine virtuel python pourrait l'exécuter. En effet, l’ordre adapté pour le faire était de remplir le fichier binaire de la façon suivante: version_pyvm, 0d0a(un hexadécimal qui se trouve dans tous les fichiers .pyc des fonctions), timestamp, marqueur de code, arg_count, local count, stack_size, flags, length_bytecode, bytecode, tuple, consts, tuple, names, tuple, varnames, tuple, freevars, tuple, cellvars, filename, name, firstlineno, length lnotab, lnotab. La plupart de l'écriture se fait par l’appel de ‘parsing_type’ pour reconnaître le type correspondant.

-T.3.2.3 :Ecriture du test ‘test-write_bitecode’ (resp. AKIKI Melissa)
(espéré 1 semaine, réel 1 semaine).

Ce test avait pour but de tester l'écriture correcte de l'exécutable python sur le fichier test ‘write_bitecode.pyc’. Ce test fait appel aux fonctions du livrable 1 notamment re_match, re_read, find_regexp. Fait appel aussi aux fonctions de remplissage de l’objet python concerné dans le livrable 2 et 3 (pyasm). En gros, cette fonction a pu tester le fonctionnement global du projet, puisqu’on a pu identifier à chaque fois si nos fonctions arriver jusqu’au bout dans leur fonctionnement. L'écriture de ce test a inspiré l’écriture de l'exécutable final du projet ./prog/pyas.c

-T.3.2.4 :Ecriture des ‘functions’ dans ‘./functions’ (resp. AKIKI Melissa)

Le but était alors de comparer avec des vrais fichiers .pyc, d'où le besoin était de créer des ‘functions’ test et comparer le binaire avec. Les ‘functions’ 1-3-5-6-7-8-9 compilent

parfaitement. Les ‘functions’ 2 et 4 concernées par le livrable 4 ne le font pas, puisque nous ne l'avons pas abordé.

-T.3.2.4 :Suppression des warning dans ‘write_bitecode.c’ (resp. AKIKI Melissa)
(espéré 1 semaine, réel 1 semaine).

La suppression des warning a été faite au fur et à mesure de l'écriture des fonctions. Notamment, les warning les plus fréquente était des warning concernant les tailles des variables mal rangées dans une variable local pour la sérialisation binaire sur le fichier binaire.

-T.3.2.4 :Libération mémoire dans ‘./prog/pyas.c’ (resp. AKIKI Melissa)
(espéré 1 semaine, réel 1 jour).

Le seule malaise qui pourrait venir quant à la libération de mémoire en ce qui concerne les fonctions de ‘write_bitecode.c’ était la fermeture du fichier *fp une fois terminé. D'où le résultat: réussi.

III Répartition des notes

Cette notation a été compliquée à mettre en place du fait que le travail UTILE est dur à quantifier, en effet il ne témoigne pas forcément des heures de travail passées sur le projet mais plutôt du fonctionnement (parfait) du code écrit.

IV Conclusion sur le projet

livrable 1: Nous sommes globalement satisfaits du livrable 1 qui marche relativement bien.

- regexp-read: fonctionne globalement bien. Certains tests d'intégration ne passent pas, pour la plupart ce sont des cas où la syntaxe est considérée comme mauvaise et pour d’autres ça n’est pas gênant car ils ne figurent pas dans la database. La mémoire n’est pas libérée en cas d’échec de parsing.
- regexp-match: satisfaisant. Les tests qui ne passent pas sont majoritairement dus à rematch-read.
- lexer: fonction qui marche, avec ou sans erreur dans le fichier .pys. Nous la testons avec les 12 fichiers .pys contenus dans le répertoire “python”. Nos tests ne sont donc pas exhaustifs. Les tests d’intégrations ne passent pas car notre bibliothèque de référence (database.txt) n’est pas faite comme attendu (en effet nous avons opté pour la spécification des opodes directement dans la database).

Livrable 2: La fonction du livrable 2 est objets.c. Son fonctionnement est correct mais encore une fois ça n’a été testé que sur une dizaines de cas. Elle n'indique que certaines erreurs (car certains “modules” sont facultatifs et donc les messages d’erreur ont été retirés dessus). Il ne manque plus grand-chose pour que la détection des fonctions se fasse correctement mais nous n’avons pas poussé plus loin car le livrable 4 ne sera de toute façon pas fonctionnel.

Livrable 3: Satisfaisant. La génération du bytecode marche avec les labels et l'écriture se fait correctement.

Livrable 4: Nous avons malheureusement étudié le livrable 4 trop tard et nous nous sommes rendus compte que pour le faire marcher il faudrait repenser entièrement le livrable 3 donc il ne sera malheureusement pas fonctionnel.

La mémoire à normalement correctement été libérée dans toutes nos fonctions et des commentaires pour comprendre leur fonctionnement ont été ajoutés.
Au niveau de la complexité nous savons qu'elle n'est pas optimale et nous avons quelques pistes d'amélioration.
Vous pouvez vous amuser à compiler les fonctions du répertoire "python" et à exécuter les .pyc créés pour toutes les fonctions sauf la 2 et 4 (livrable 4).

Nous sommes globalement satisfaits du rendu du projet.

Annexe 1 : Tableau de répartitions des fonctions

MODULE	FONCTIONS	ECRITURE	CORRECTION	TEST
char_group.c	char_group* new_char_group()	R		L
	void char_group_initialize(char_group *chargroup)	R		L
	int delete_char_group (void *cg)	R		L
	int char_group_print(void *cgx)	R		L
database.c	data_t data_new(char *type, char *def);	L R		L
	int data_delete(void* data);	R		L
	int print_data_t(void * dat);	R		L
	queue_t read_database(char* nom_fichier_ass);	R M		L/R
config/pyas.conf		M		
grammar.c	int parse_pys (list_t * lexems);	L/R		L
	int parse_prologue (list_t * lexems);	L/R		L
	int parse_set_version_pyvm (list_t * lexems);	L/R		L
	int parse_set_flags (list_t * lexems);	L/R		L
	int parse_set_filename (list_t * lexems);	L/R		L
	int parse_set_name (list_t * lexems);	L/R		L
	int parse_interned_strings (list_t * lexems);	L/R		L
	int parse_constants (list_t * lexems);	L/R		L
	int parse_constant (list_t * lexems);	L/R		L
	int parse_list (list_t * lexems);	L/R		L
	int parse_tuple (list_t * lexems);	L/R		L
	int parse_names (list_t * lexems);	L/R		L
	int parse_varnames (list_t * lexems);	L/R		L
	int parse_freevars (list_t * lexems);	L/R		L
	int parse_cellvars (list_t * lexems);	L/R		L
	int parse_code (list_t * lexems);	L/R		L
	int parse_assembly_line (list_t * lexems);	L/R		L
	int parse_label (list_t * lexems);	L/R		L
	int parse_source_lineno (list_t * lexems);	L/R		L
	int parse_insn (list_t * lexems);	L/R		L
	int parse_eol (list_t * lexems);	L/R		L
	int parse_set_directives (list_t * lexems);	L/R		L
	int parse_set_source_size (list_t * lexems);	L/R		L
	int parse_set_stack_size (list_t * lexems);	L/R		L
	int parse_set_arg_count (list_t * lexems);	L/R		L
	int parse_set_kwonly_arg_count (list_t * lexems);	L/R		L
	int parse_set_posonly_arg_count (list_t * lexems);	L/R		L

lexem.c	lexem_t lexem_new(char *type, char *value, int line, int column);	L		L
	int lexem_print(void *_lex);	L		L
	int lexem_delete(void *_lex);	L		L
	lexem_t lexem_peek(list_t *lexems);	L	R	L
	lexem_t lexem_peek_pass_new_line(list_t *lexems);	L		L
	int lexem_compare(lexem_t lex1, lexem_t lex2);	L		L
	int lexem_compare_also_column_and_line(lexem_t lex1, lexem_t lex2);	L		L
	int lexem_type (lexem_t lex , char * type);	L		L
	lexem_t lexem_advance(list_t *lexems);	L	R	L
	int next_lexem_is(list_t *lexems, char *type);	L		L
	int print_parse_error(char *msg, list_t lexems);	L		L
lexer.c	void init_word (char* word);	M R		
	list_t find_regexp(char* name_data_base, char* file_name);	M R		
list.c	int list_compare_lexem(list_t l1, list_t l2);	L		L
	list_t list_add_last(list_t l, void* object);	L		L
	int list_print_first(list_t l, action_t print);	L		L
objet.c	pyobj_t pyobj_new();	R		R
	void pyobj_delete(pyobj_t pyobj);	R	L	R
	py_codeblock* codeblock_new();	R		R
	void codeblock_delete(py_codeblock* codeblock);	R		R
	void print_codeblock (py_codeblock* codeblock);	R		R
	void print_pyobj (pyobj_t pyobj);	R	L	R
	void string_print (void* string);	R		R
	int fill_pyobj_list (pyobj_t pyobj_list, list_t* list, list_t* type);	R		R
	int char_delete (void* char_);	R		R
	int char_print_2 (void* test);	R		R
	int fill_timestamp (py_codeblock* codeblock);	R		R
	char* str_sans_guillemets (char* string);	R		R
	int fill_pys (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_prologue (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_version_pyvm (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_flags (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_filename (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_name (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_interned_strings (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_constants (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_constant (list_t * lexems, list_t* constant_list, list_t* type);	R		R
	int fill_list (list_t * lexems, list_t* constant_list, list_t* type);	R		R
	int fill_tuple (list_t * lexems, list_t* constant_list, list_t* type);	R		R
	int fill_names (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_varnames (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_freevars (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_cellvars (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_code (list_t * lexems);	R		R
	int fill_assembly_line (list_t * lexems);	R		R
	int fill_label (list_t * lexems);	R		R
	int fill_source_lineno (list_t * lexems);	R		R
	int fill_insn (list_t * lexems);	R		R
	int fill_eol (list_t * lexems);	R		R
	int fill_set_directives (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_source_size (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_stack_size (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_arg_count (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_kwonly_arg_count (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_set_posonly_arg_count (list_t * lexems, py_codeblock* codeblock);	R		R
	int fill_function (list_t * lexems, py_codeblock* codeblock);	R		R
production_bytecode.c	label_address_t label_address_new (char* name, int address, int address2);	L/R		L
	int label_address_delete (void* label1);	L		L
	int label_address_print (void* label1);	L		L
	unsigned char opcode_is(lexem_t lex);	L		L
	unsigned char * opcode_is_char(lexem_t lex);	L		L
	int argument_yes_or_no(lexem_t lex);	L		L
	pyobj_t pyobj_new_string(int len);	L		L
	int length_calculated_for_bytelist(list_t l);	L		L
	int lenght_lnotab(list_t lexem);	L		L
	pyobj_t pyobj_from_bytecode_list(list_t l);	L		L
	pyobj_t pyobj_from_lnotab_tab(char* lnotab, int taille);	L		L
	void find_the_part_instruction(list_t* lexem);	L	R	L
	int pyasm(pyobj_t code, list_t lexem);	L		L
queue.c	queue_t queue_new(void);	L		L
	int queue_empty(queue_t q);	L		L
	queue_t enqueue(queue_t q, void* object);	L		L
	list_t queue_to_list(queue_t q);	L	R/M	L
	void* queue_peek(queue_t f);	L		L

	queue_t dequeue(queue_t q);	L	R	L
	int queue_print(queue_t q, action_t print);	L		L
	int char_print(void * c);	L		L
	int queue_length(queue_t q);	L		L
	int queue_visit(queue_t l, action_t action);	L		L
	queue_t queue_next(queue_t q);	L		L
	void queue_delete(queue_t l, action_t delete);	L	R	L
re_parse_livrable_1.c	int re_read_2(char *data_in, queue_t* queue);	R		R
	int read_crochet_2(char *data_in, char_group *chargroup);	R		R
	queue_t fillchargroup_2 (queue_t queue,char_group *chargroup);	R		R
re_parse.c	queue_t re_read(char *data_in);	R		R
	void read_crochet(char *data_in, char_group *chargroup);	R	M	R
	queue_t fillchargroup (queue_t queue,char_group *chargroup);	R	M	R
regexp_cg.c	int re_match(char *regexp , char *source , char **end);//fonction initiale readapter pour lire des char_group	M		M
	int re_match_cg(queue_t regexp, char *source, char **end);//fonction pour comparer des char_group avec la source	M		M
	char *asci(int cg[256]);	M		M
regexp.c	int re_match(char *regexp , char *source , char **end);	M		M
write_bytecode.c	void parsing_type(FILE *fp, pyobj_t obj);	M		M
	int pyobj_write(FILE *fp, pyobj_t obj);	M		M
Test d'intégration		M		
Création des éxécutables finaux		M	R	