Compte rendu : livrable 3

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 et surtout le livrable 3.

I Readme

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

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

#data_base.txt
But: fichier regroupant tous les types avec les definitions associées; c'est le dictionnaire complet de lexem

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

#queue.c
But: Définir toutes les fonctions nécéssaire 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, enqueue, queue_to_list, queue_peek, dequeue, queue_length.

#list.c
But: Définir toutes les fonctions nécéssaire 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 "
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 nécéssaire à l'utilisation des chargroup

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 data_base.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-database.exe data_base.txt.test " , pour tester seulement un échantillon. Sinon " ./bin/unit/test-database2.exe data_base.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-regexp-cg.exe [une expression reguliere] [une source a comparer avec]". exemple [a-c]+x aaaax

Pour tester avec les test d'integrations : 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   data_base.txt.test ".

le programme va prendre le fichier data_base.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 data_base.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 ".

#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: Non fonctionnel

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 la fonction pyasm : Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-production_bytecoe.exe -v" EN attente pas testée

#write_bitecode.c

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

Avancement: Fonctions necessaires et test correspondant ecrit, fonctionnel-presque (pas d'erreur, ni segfault).

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" , un message de 'write to file ok'. Ensuite aller voir le fichier text "write_bitecode.txt" de l'executable ecrit en binaire

II Répartition des tâches

A) LIVRABLE 1

— T.1 "Parsing des expressions régulières" : Finir le livrable 1.

— T.1.1 "Correction regexp_cg" (resp. AKIKI Melissa) :. (espéré 1 semaine, réel 3 semaines);
“regexp_cg" est à présent adapté aux nouvelles définitions, sais lire à présent le caractère ‘\n’ et ‘\t’ d’une façon correcte, lors de son appel dans ‘lexer’.
Regexp a été modifié (adaptations de conditions d'arrêts, gération d’un segfault d’un cas très particulier non perçu avant, élimination d’une fonction intermédiaire ‘re_one’, et son écriture d’une façon plus simple et intuitive.) Regexp en totale a été modifié pour passer plus de tests d’intégrations. Il reste à régler les cas des hexadécimaux qu’il reconnaît comme des float et les newlines qui ne sont pas reconnues après les float. Et ce problème a été réglé.

— T.1.1.1 "Réalisation des tests d’intégration regexp_cg" (resp.AKIKI Melissa)
les tests ’02_test_regexp_match’ fonctionnent.(53/65)

— T.1.2 "Correction re-parse" (resp. DUCHADEAU Romain) : Fonctionnel (espéré 1h).

— T.1.3 "Correction lexer" (resp. DUCHADEAU Romain) : Lexer est totalement fonctionnel.

— T.1.4 "Correction test-database.c" (resp. Laura) : Réalisation du test avec unitest sur un échantillon de database.txt (espéré 15 min, réel 1h); qui ne fonctionne plus car database.txt.test a été modifié.

— T.1.5 Ecriture des exécutables des livrables 1 et 2 (resp. DUCHADEAU Romain et AKIKI Melissa) : (espéré 30 min , reel 1 heure). *note: Pour l'instant, les exécutables sont dans le répertoire ./bin/unit/ ; ils sont nommés livrable_x_xxx.exe . Ils seront déplacés quand tout sera parfaitement fonctionnel dans un répertoire dédié)*

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 "Correction des fonctions utiles" : Fonctions pour gérer les lexem, dans les fonctions de grammaires

— T.2.1.1 "Correction de lexem.c" (resp. RASCOL Laura & DUCHADEAU Romain) : lexem_peek, lexem_type, lexem_advance, next_lexem_is, print_parse_error. (espéré 1 jour, réel 1 jour)
— T.2.2.2 "Test des fonctions de lexem.c et libération de la mémoire dans la fonction test" (resp. RASCOL Laura) : Tests réalisés avec unitest, cette étape fut compliqué dans le sens où savoir où libérer la mémoire n'était pas chose facile.

— T.2.2 "Ecriture des fonctions pour chaque non-terminal" : Fonctions qui vérifient la syntaxe en renvoyant 1 ou 0

— T.2.2.1 "Correction de grammar.c" (resp. RASCOL Laura et DUCHADEAU Romain) : 29 fonctions du sujet (voir grammar.h). (espéré 1jour, réel 4 jours).
— T.2.2.2 "Ajouter des tests pour tester des fonctions de grammar.c" (resp. RASCOL Laura) : Tests relativement exhaustifs réalisés avec unitest (79 tests ont été mis en place) (espéré 4 jours, réel 1 semaine)

— T.2.3 "Ecriture de la fonction qui remplit la structure python" : Construction d'un objet de code python

— T.2.3.1 "Ecriture de objet.c" (resp. DUCHADEAU Romain) : . (espéré 1 semaine, réel 1 semaine)
— T.2.3.1.1 "Ecriture de pyobj_new" : créer et allouer un nouvel objet python
— T.2.3.1.2 "Ecriture de codeblock_new" : créer et allouer un nouveau py_codeblock
— T.2.3.1.3 "Ecriture de pyobj_delete" : supprimer un objet python et libérer la mémoire
— T.2.3.1.4 "Ecriture de codeblock_delete" : supprimer py_codeblock et libérer la mémoire
— T.2.3.1.5 "Ecriture de print_pyobj " : imprimer un pyobj sur le terminal
— T.2.3.1.6 "Ecriture de print_codeblock " : imprimer un codeblcok sur le terminal
— T.2.3.1.7 "Ecriture de fill_pyobj_list " : remplir un pyobj de type list a partir d'une liste de "caractère" et d'une liste contenant leurs type
— T.2.3.1.8 "Ecriture des fonctions correspondant à la grammaire " : ce sont ces fonctions qui constituent le corps principal du livrable 2.

— T.2.3.3 "Test de objet.c" (resp. DUCHADEAU Romain) : (espéré 1h, réel qques h), Puisque les tests de grammar.c ont été exhaustifs, et que objet.c s'en inspire beaucoup dans sa structure (il remplit simplement un objet python en même temps que de vérifier la

syntaxe), nous avons fait le choix de tester (d'afficher) seulement pour la fonction globale qui est fill_pys et vérifier à l'oeil que tout est bien rempli, un test_oracle aurait été trop compliqué à mettre en place au vu de la longueur de l'affichage.

C)LIVRABLE 3

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 non terminé).

T.3.1.1 "Récupérer le opcode et l'argument dans le type de insn″ (resp. RASCOL LAURA) (espéré 1 jour, réel 3 jours) :

— T.3.1.1.1 "Ecriture de la fonction opcode_is" (resp. RASCOL LAURA) : . (espéré 1 jour, réel 4 jours).
— T.3.1.1.2 "Ecriture du test de la fonction opcode_is" (resp. RASCOL LAURA) : (espéré 1 jour, réel 1 jour) test dans le test-opcode-is.c
— T.3.1.1.3 "Ecriture de la fonction argument_yes_or_no" (resp. RASCOL LAURA) : . (espéré 1 jour, réel 4 jours).
— T.3.1.1.4 "Ecriture du test de la fonction argument_yes_or_no" (resp. RASCOL LAURA) : (espéré 1 jour, réel 1 jour) test dans le test-argument-yes-or-no

---

T.3.1.2 "Transformer une liste en string″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 3 jours) en effet nous avons fait le choix de ranger les informations de bytecode dans une liste, pour un souci d'allocation, mais pour les ranger dans l'objet python il faut les convertir en string dans un second temps.

— T.3.1.2.1 "Écriture de la fonction pyobj_t pyobj_from_bytecode_list(list_t l) " (resp. RASCOL LAURA) : . (espéré 1 jour, réel1 semaine) Cette fonction permet de produire le bytecode à partir d'une liste de lexem (opcode et integer). Après plusieurs changement d'avis au lieu de prendre en argument une liste d'unsigned char, il est plus simple de lui mettre une liste de lexem, et de faire les modification (pour avoir des char finalement dans le pyobj à la fin).
— T.3.1.2.2.1 "Ecriture de la fonction pyobj_t pyobj_new_string(unsigned len)" (resp. RASCOL LAURA) : (espéré 1 jour, réel 2 jours) Fonction utile aussi pour allouer un lnotab. Elle permet d'allouer dynamiquement l'espace du py obj qui sera utile pour stocker les informations, et donner le type PYOBJ_CODE.
— T.3.1.2.2.2 "/ int length_calculated_for_bytelist(list_t l)" (resp. RASCOL LAURA) : (espéré 1 jour, réel 4 jours) Cette fonction permet de calculer la taille nécessaire à allouer dan sle pyob pour le bytecode.

— T.3.1.2.2.3”Ecriture de **test-length-calculated-for-bytelist.c** ” : elle permet de tester / int length_calculated_for_bytelist(list_t l) avec unitest.
— T.3.1.2.3 “**test-pyobj-from-bytecode-list.c** ” : elle permet de tester la transformation d’une liste de lexème alternant instruction et argument (si il y’a) en la transformant en pyobj via des printf.

T.3.1.3 "Transformer un tableau en string″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 3 jours) Finalement le choix pour ranger le lnotab a été un tableau dont la taille est calculé au préalable.

— T.3.1.3.1 "Écriture de la fonction pyobj_t pyobj_from_lnotab_tab(char* lnotab, int taille) ” (resp. RASCOL LAURA) : . (espéré 1 jour, réel 1 semaine) Après plusieurs changement d’avis au lieu de prendre en argument une liste d’int, finalement il est préférable (en terme de mémoire et de rapidité de calculer la taille du tableau et de l’allouer directement.
— T.3.1.3.2.1 "Écriture de la fonction int lenght_lnotab(list_t lexem) ” : fonction qui permet de calculer la taille du tableau de lnotab.
— T.3.1.3.2.2 "Écriture de la fonction **test-lenght-lnotab.c** ” : fonction qui permet de tester int lenght_lnotab(list_t lexem) avec unitest.
— T.3.1.3.3 "Écriture de la fonction **test-pyobj-from-lnotab-tab.c** ” : fonction qui permet de tester si le pyob est bien rempli à partir d’un tableau de char.
(— T.3.1.2.1” Ecriture des fonctions list_t list_del_first_without_action(list_t l) et list_t list_delete_without_action(list_t l); pour supprimer une liste d’entier nécessaire au début quand le choix était de stocker lnotab dan sune liste d’entier.”)

T.3.1.4 "Récupérer le bon pointeur sur la partie instruction à partir de la liste de lexem″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 2 jours).

— T.3.1.4.1 “Ecriture de la fonction list_t * find_the_part_instruction(list_t lexem)” (resp. RASCOL LAURA) : . (espéré 1 jour, réel 4 jours).
— T.3.1.4.2 “Ecriture du test de la fonction list_t * find_the_part_instruction(list_t lexem) ” (resp. RASCOL LAURA) : (espéré 1 jour, réel 1 jour)

T.3.1.5 “Ecrire la fonction globale″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel pas fini).
T.3.1.5.2 “Tester la fonction pyasm de manière exhaustive″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel pas fait).
— T.3.1.5.1 “Ecrire la fonction lexem_compare ″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 2 jours).
— T.3.1.5.2 “Tester la fonction lexem_compare ″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 1 jour).

— T.3.1.5.3 “Ecrire la fonction list_compare_lexem ″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 4 jours).
— T.3.1.5.4 “Ecrire la fonction list_compare_lexem ″ (resp. RASCOL LAURA) : . (espéré 1 jour, réel 2 jours).

T.3.2 “ Ecrire dans un fichier texte en binaire(resp. AKIKI Melissa) (fonction pyobj_write(FILE *fp, pyobj_t obj)”)

T.3.1.1 “write_bitecode.c″ (resp. AKIKI Melissa) : (espéré 2 jours, réel 5 jours).
— T.3.1.1.1 “Ecriture de la fonction parsing_type” (resp. AKIKI Melissa) : . (espéré 1 jour, réel 4 jours).
Fonction qui écrit le type correspondant de l'objet python en binaire sur le fichier texte.
— T.3.1.1.2 “pyobj_write” (resp. AKIKI Melissa) : (espéré 1 jour, réel 4 jour)
Écriture de la fonction finale, qui devra écrire sur un fichier texte en binaire l'exécutable de la machine python. Cette fonction appellera la fonction ‘parsing_type’
— T.3.1.1.3 “test-write_bitecode.c” (resp. AKIKI Melissa) : (espéré 1 jour, réel 1 jour)
Remplissage objet python en appelant fonctions convenable du livrable 2 , ensuite appel de fonction pyobj_write qui écrira en binaire sur un fichier ‘write_bitecode.txt’.
Test correspondant pour tester ‘pyobj_write’ fonctionnel presque (pas d’erreur, ni segfault), en cours de test. Manque de s’assurer de la bonne écriture binaire du fichier.

D)Livrable 4

—T.4 “Pouvoir traiter des fonctions l’objet Python” Pas encore traité (ici le livrable 3 ne fonctionne pas encore)
—T.4.1 “Adapter makefile pour les éxécutables” : Ceci comprends de bien les nommer de les positionner dans un autre dossier, et les faire compiler.

—T.4.2 “Modifier objet.c”
—T.4.3 “Tester modif objet.c”
—T.4.4 “Modifier grammar.c”
—T.4.5 “Tester modif grammar.c”