RASCOL Laura | AKIKI Melissa | DUCHADEAU Romain

Compte rendu : livrable 2

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 et le livrable 2.

I Justification de nos choix

A) structure data_t pour le livrable 1

Après avoir écrit la database.txt, le but a été de stocker le type ainsi que la définition, afin de pouvoir les comparer au lexem. Pour nous, il était inutile de stocker ces deux informations dans une structure lexem_t, car la valeur de la ligne et de la colonne ne nous importait pas. Afin d'être plus efficace, nous avons donc créé la structure data_t.

B) Structure chargroup

Le premier choix était de stocker les caractères dans des char*, cette solution était faisable mais très peu pratique (la gestion des chra* étant parfois compliquée) et surtout très lourde en mémoire. Nous avons donc opté pour un tableau d'entier de 256 caractères dans lequel nous mettons 0 si le caractère ne figure pas dans la liste de possibilités et 1 si il y figure. La encore le choix du int n'est pas optimal niveau place mémoire, nous opterons dans le futur pour un unsigned char qui prend moins de place.

C) Regexp

L'écriture de regexp à été faite en plusieurs étapes, c'est à dire que ce module a été premièrement traiter pour lire des regexp de type char*, ensuite adapté pour lire des regexp de type chargroup. Dans un dernier temps, il a été une nouvelle fois réécrit et réadapté afin

de pouvoir renvoyer ce qu'on veut pour faciliter l' écriture de lexer. Cette fonction s'occupe de s'assurer que la source match bien avec la définition.

D) Parse

Le but de parse.c est de pouvoir lire un char* (représentant une expression régulière) et la transformer en une queue de chargroup qui sera ensuite lisible par re_match_cg . C'est cette fonction qui permet de passer d'une expression "mathématiques" des choses à quelque chose de compréhensible par notre algorithme.

E) Lexer

Lexer avait des erreurs de compilations et des "segmentations faults" lors de la première écriture. A présent réécrite avec un nouvel algorithme, qui range la database.txt dans une liste de data_t. Ensuite on a mis en place des boucles pour lire lignes par lignes le fichier .pys et comparer chaque mot avec la database. On appellera la fonction re_read pour lire la définition et rendre une queue de chargroup. Ensuite nous allons appeler re_match_cg qui va lire la source et la comparera avec notre liste de regexp ainsi rendu par re_read a l'aide de la fonction queue_to_list. Un problème d'allouage de queue(qui est une queue temporaire qu'on veut ré-allouer à chaque fois) cause que la fonction arrive à lire la première ligne du fichier .pys et tombe en segmentation fault.

II Readme

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

#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.

RASCOL Laura | AKIKI Melissa | DUCHADEAU Romain

#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 "

#char_group.c

But: Définir toutes les fonctions nécéssaires 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 ".

#regexp.c

But: Comparer une source char* avec une expression régulière

Avancement: 3 tests sur 5 fonctionnel (mais régalges de ces problème sur la version adaptée à nps choix)

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 sauf pour '^'

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

#parse.c

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

Avancement: Fonctionne

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande " ./bin/unit/test-re-parse.exe text_a_trier ".
le programme va prendre le text_a_trier et mettre les différents 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: compile et fonctionne bien pour un seul rebouclement de loop, apres segmentation fault(probleme de reallouage)

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/test_function.pys ".

#lexem.c

But: Fonctions utiles pour grammar.c

Avancement: Fonctionnel

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

#grammar.c

But: Fonctions pour chaque non-terminal : Fonctions qui vérifient la syntaxe en renvoyant 1 ou 0

Avancement: Non fonctionnel

Pour tester: Tapez d'abord "make" dans le terminal pour compiler, Tapez ensuite la commande "./bin/unit/test-grammar.exe " (la fonction test-grammar2.exe était utile pour débugger).

#objet.c

But: Fonction qui remplit la structure python : Construction d'un objet de code python

Avancement: Non fonctionnel

III Répartition des tâches

A) Finalisation livrable 1

Ici vous retrouverez, les tâches et sous-tâches permettant de valider l'analyse lexicale demandé pour le 04 Octobre 2022 (livrable 1, faites pendant ces deux dernières semaines).

— T.1 "Parsing des expressions régulières" : Découpage d'une expression régulière en la liste de ses composants ; prendre en main le sujet, le bootstrap fournit pour démarrer et les outils fournis dans ce bootstrap pour soutenir les tests.

— T.1.2 "Finalisation de la version adaptée regexp_cg.c" (resp. AKIKI Melissa) (espéré 1 semaine, réel 3 semaines).

— T.1.2.3 "Fonction rematch_cg" (resp.AKIKI Melissa)
Fonction qui prend une source (type char*) et une expression régulière sous forme de liste, fournis par re-parse. et selon le type d'opérateur appelle la fonction correspondante de traitement.(les cas 'NULL' et " " ont été pris en considération).
A noter que regexp est une fonction nécessaire pour d'autres fonctions, notamment dans 'lexer.c" lui sera appelé pour lire le fichier .pys.
— T.1.2.3.1 "Fonction re_match_zero_or_more" (resp.AKIKI Melissa)
Fonction qui s'assure de la bonne syntaxe au cas d'une '*'.c-a-d s'assure que l'expression régulière de type char_group figure dans la source zéro ou plusieurs fois.
— T.1.2.3.2 "Fonction re_match_one_or_more" (resp.AKIKI Melissa)
Fonction qui s'assure de la bonne syntaxe au cas d'un '+'.c-a-d s'assure que l'expression régulière type char_group figure dans la source au moins une fois.
— T.1.2.3.3 "Fonction re_match_zero_or_one" (resp.AKIKI Melissa)
Fonction qui s'assure de la bonne syntaxe au cas d'un '?'.c-a-d s'assure que l'expression régulière type char_group figure dans la source une fois ou aucune.
— T.1.2.3.4 "Fonction re_match_negation" (resp.AKIKI Melissa)
Fonction qui traite le complément de l'expression régulière type char_group si elle est présente dans la source.
— T.1.2.3.5 "Fonction re_one" (resp.AKIKI Melissa)
Fonction qui s'assure de la bonne syntaxe au cas d'un seul caractère .c-a-d s'assure que l'expression régulière type char_group figure dans la source au moins une fois..
— T.1.2.3.6 "Fonction is_equal" (resp.AKIKI Melissa)
Fonction intermédiaire à qui toutes les autre rematch vous appeler pour pouvoir tester l'égalité entre deux char*.(pris en considération le caractère '.').
— T.1.2.4 "Test de regexp_cg" (resp.AKIKI Melissa) Analyse des tests unitaires Fournis dans tests/unit/test-regexp_cg.c et des tests d'intégration fournis dans le répertoire 01_test_regexp_parse. Cette fonction permet de tester des expressions intéressantes (notamment des cas particuliers).

— T.1.5 "Changement de chargroup.c" (resp. DUCHADEAU Romain) : Choix de la structure la plus adapté, pour stocker les informations et les fonction associées à cette structure.(espéré 1 jour, réel 3 jours, le temps de réfléchir à la méthode la plus adaptée et de tester les possibilités)

— T.1.5.1 "Créer la structure la plus efficace" (resp. DUCHADEAU Romain) : Le but était de stocker à la fois les valeurs admises et leur occurrence sous une forme lisible par regexp.
— T.1.5.2 "Ecrire les fonctions new_char_group, delete_char_group, char_group_print" (resp. DUCHADEAU Romain) (espéré quelques heures, effectif: quelques heures)
— T.1.5.1 "Tester les fonctions : new_char_group, delete_char_group, char_group_print " (resp. RASCOL Laura) : Tests réalisés avec unitest;

— T.1.3 "Ecriture de parse.c" : Permet de lire un data_t et de rendre sa définition lisible par un algorithme comme re_match
(espéré 1 semaine, réel 3 semaines).

— T.1.3.1 “Ecriture de read_crochet” (resp.DUCHADEAU Romain): le but de cette fonction est de lire l’intérieur des crochets et de stocker ces valeurs sous forme de char group, elle est utilisée dans la fonction re_read.
— T.1.3.2 “Ecriture de re_read” (resp. AKIKI Melissa DUCHADEAU Romain): le but de cette fonction est de lire un char* et de stocker les valeurs “une à unes” dans une liste de char groupe. Réglage de quelques ‘segmentation fault’ et de problèmes de réallocations (resp. AKIKI Melissa)
— T.1.3.3 “Ecriture de “fillchargroup” (resp. AKIKI Melissa): but d'éliminer les bouts de code copier-coller dans re_read qui peut être exhaustif et le plus important, cette fonction a été écrite pour éliminer tous problèmes de réallocation de pointeurs dans re_read.
— T.1.5.1 "Tester les fonctions” : (resp. DUCHADEAU Romain).

— T.1.3 "Ecriture de lexer.c" : Fonction finale du livrable 1
(espéré 1 semaine, pas fini).-en cours de travail (en fait lexer fonctionne bien pour un seul rebouclement et puis fait un segmentation fault)
— T.1.3.1 “Ecriture de find_regexp” (resp. AKIKI Melissa et DUCHADEAU Romain): le but de cette fonction est de lire un fichier assembleur et de renvoyer une liste de lexem. En soit c’est la fonction finale. En effet, cette fonction lit le database.txt et le range dans une liste de data_t, et pour chaque ligne de code .pys, compare avec la database en appelant reparse, pour transformer la définition, puis queue_to_liste, et appelle rematch_cg qui ne peut que lire des listes pour s’assurer de la bonne syntaxe. Au cas où c’est bon, on range alors le type, la valeur, la ligne et la colonne du mot en question dans un lexem. Sinon, on passe a la definition prochaine. Au cas où, on a parcouru tous le fichier .txt de la database et aucune ne correspond au mot écrit, on imprime un message de “synthaxe error”.

B) Livrable 2

Ici vous retrouverez, les tâches et sous tâches permettant de développer un parser, demandé pour le 18 Octobre 2022 (livrable 2, faites pendant ces deux dernières semaines).

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

— T.2.1.1 "Ecriture de lexem.s" (resp. RASCOL Laura) : lexem_peek, lexem_type, lexem_advance, next_lexem_is, print_parse_error. (espéré 3 jours, réel 1 semaine)
— T.2.2.2 "Test des fonctions de lexem.c" (resp. RASCOL Laura) : Tests réalisés avec unitest;

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

— T.2.2.1 "Ecriture de grammar.s" (resp. RASCOL Laura et DUCHADEAU Romain) : 29 fonctions du sujet (voir grammar.h). (espéré 1 semaine, réel 1 semaine)h
— T.2.2.2 "Test des fonctions de grammar.c" (resp. RASCOL Laura) : Tests (pour l’instant non exhaustifs) réalisés avec unitest;

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

— T.2.2.1 "Ecriture de grammar.s" (resp. RASCOL Laura et DUCHADEAU Romain) : 29 fonctions du sujet (voir grammar.h). (espéré 1 semaine, réel 1 semaine et +)
— T.2.2.2 "Test des fonctions de grammar.c" (resp. RASCOL Laura) : Tests (pour l’instant non exhaustifs) réalisés avec unitest;

— 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.s" (resp. DUCHADEAU Romain) : . (espéré 1 semaine, rée lpas possible car il était nécéssaire que grammar.c fonctionne)

III État des lieux - Conclusion

Le livrable 1 est presque terminé.

Pour ce qui est du livrable 2, il est bien commencé, mais il reste la partie où il faut ranger les objets python.