Mini-projet C99 : Interpréteur *PostFixe* *ver. 1.0*

Le projet est présenté en version basique puis des extensions sont proposées. Le travail demandé est indiqué ensuite. À la fin du document se trouvent quelques conseils et mises en garde.

Les fichiers, les mises à jour et une foire aux questions seront disponibles sous Celene.

Le but est de réaliser un interpréteur pour le langage *PostFixe*. Ce langage est post-fixé, c'est-à-dire que l'opérateur arrive après les opérandes. Typiquement, une addition s'écrit `5 3 +` et une exécution conditionnelle `<block> <cnd> if`. Cela peut sembler inhabituel comme façon de programmer, mais il n'y a pas plus de syntaxe, aucune priorité entre les opérateurs... La lecture du code est très simple [1].

Les programmes en *PostFixe* ont l'extension `.pf`. Des exemples sont fournis et il ne s'agit pas d'en écrire d'autres. L'interpréteur à réaliser en C99 fonctionne de la façon suivante :

pf `<options> [<fichier>.pf]`

S'il n'y a pas de fichier en entrée, le programme est lu sur l'entrée standard. Comme option il y a `-h` pour obtenir de l'aide et `-t` pour avoir une trace [2] de l'interprétation.

L'interpréteur fonctionne comme décrit sur la Fig. 1 : tant qu'il y a des `chunk` à lire sur le flux d'entrée, l'interpréteur lit le premier et agit sur la pile et le dictionnaire (tableau associatif, voir documentation) en fonction de celui-ci :

— s'il s'agit d'une valeur (`value`), celle-ci est empilée sur la pile (`stack`).
— s'il s'agit d'un opérateur (`operator`), celui-ci est *déclenché*. Selon sa nature, il agit sur la pile, le dictionnaire... et même l'interpréteur pour une exécution conditionnelle ou une boucle.

Quand le flux est vide, l'interpréteur affiche le contenu de la pile et s'arrête.

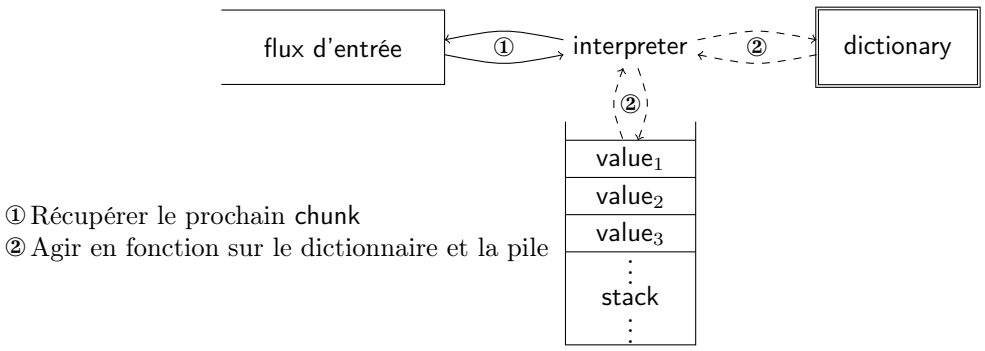

FIGURE 1 – Schéma de fonctionnement.

Pour simplifier :

— il n'y a pas d'opérateur d'entrée (mais il y a quand même des opérateurs de sortie), et
— il n'y a pas de commentaires dans le langage *PostFixe*.

1 Version basique

Il faut implanter tous les modules de l'archive (compléter les `.c`). À chaque fois les tests correspondants doivent passer sans erreur (pour les lancer tous `make test`). Les fichiers entête sont documentés et la documentation en `html` est fournie. Il est préconisé de réaliser le projet dans l'ordre suivant, de bien lire la documentation et de réfléchir avant de coder.

1.1 Modules utilitaires

Les modules suivants sont fournis :

— `basic_type` servant à stocker tout type base (comme `answer` du cours, mais avec des modifications), il est considéré comme un type simple et n'est jamais manipulé par pointeur, et

1. C'est la base du langage postscript interprété pas des imprimantes depuis plus de 30 ans !
2. Attention, la trace est là pour aider, il faut regarder les fichiers de sortie avec trace et respecter le format car elle est vérifiée par les tests.

— chunk servant de base à une approche orienté objet (comme agent du cours, mais avec des modifications), il est toujours manipulé par pointeur.

Il faut implanter les modules suivants :

— sstring à tester avec make t_sstring,
— linked_list_chunk à tester avec t_linked_list_chunk, et
— dictionary à tester avec t_dictionary.

1.2 De value_int jusqu'à l'interpréteur

Il s'agit d'écrire le module de ce type simple puis de mettre en place tout l'interpréteur afin de comprendre le fonctionnement et l'agencement des modules. Ensuite il *ne reste plus* qu'à ajouter, un à un, les types puis les opérateurs en modifiant ce qui doit l'être.

Une fois le module value_int complété et testé avec t_value_int, il faut commencer le module read_chunk_io afin de pouvoir lire un entier. Un entier commence par le signe '-' ou par un chiffre (mais pas par '+'). La suite est gérée par une fonction à part (à modifier plus tard pour reconnaître également les double).

Les modules interpreter et pf sont aussi à commencer. Le test TV01 doit passer.

1.3 Toutes les value

Les codes des value se ressemblent beaucoup. Il est conseiller de partir d'une version parfaite de value_int et d'en tirer des patrons (avec #define, stockés dans le fichier macro_value_c.h) pour les différentes parties des .c. Les .h sont déjà écrits de cette façon.

Pour tester au fur et à mesure sur des programmes particuliers utiliser make TV## où ## est le numéro du programme dans DATA/Programs/prog_v_*.pf. L'ordre suivant est préconisé :

— value_double dès qu'il y a un '.', ce n'est plus un entier (TV02) ;
— value_boolean correspond aux mots-clés true et false (TV03) ;
— value_protected_label commence par '\', puis une lettre et une suite de lettres, chiffres et '_' ; cette suite ne doit pas être un mot-clé [3] (TV04) ;
— value_sstring commence par '"' (TV05) ; et
— value_block commence par '{' et peut contenir des sous-blocks et des opérateurs (qui ne sont pas déclenchés) (TV06),

Utiliser make t_value pour les tester toutes.

Le module value_error est donné en entier. Une erreur ne peut être modifiée (*immuable*) ce qui permet de faire des copies simples et sûres. Expliquer dans le **compte-rendu** le fonctionnement du mécanisme de copie (et la gestion de la mémoire attenante). Dire où ce schéma est utilisé/utilisable de nouveau et là où il ne peut l'être.

Sachant qu'il n'y a que deux valeurs possible pour les booléens, proposer un autre schéma pour les créations et copies qui ne fassent ni malloc ni free. L'expliquer dans le **compte-rendu**.

1.4 Tous les operator

Les codes des operator se ressemblent beaucoup... #define... fichier macro_operator_c.h...

Pour tester au fur et à mesure sur des programmes particuliers make TO## (prog_o_*.pf).

Le fonctionnement des opérateurs est décrit dans la documentation et le .h. Il est conseillé de faire les modules (et tests) dans l'ordre suivant :

— operator_nop ne fait rien (TO01) ;
— operator_pop détruit la valeur en haut de pile (TO02) ;
— operator_print et operator_print_stack (TO03) ;
— operator_stop_trace et operator_start_trace (TO04) ;
— operator_copy (TO05) ;
— operator_addition (TO06), puis en en faisant un modèle : operator_subtraction, operator_division et operator_multiplication puis operator_remainder (TO07) ;
— operator_not (TO08), puis operator_and et operator_or (TO09) ;
— operator_equal et operator_different (TO10) ;

3. Ceux-ci sont listés dans le module operator_creator_list.

— operator_less et operator_less_equal (TO11) ;
— operator_def (TO12), puis operator_print_dictionary (TO13), puis operator_label (TO14) ;
— operator_if (TO15), puis operator_if_else (TO16) ; et
— operator_while (TO17).

2 Extensions

Il n'y a pas de programmes pour tester les extensions, ceux-ci doivent donc être fournis, ainsi que la réponse attendue et les éventuels fichiers annexes pour les faire fonctionner.

2.1 Opérateurs de value_sstring

Il s'agit d'implanter les opérateurs suivants :
— sstring_protect retourne une version protégée d'une chaîne : "\"" au lieu de '"', "\n"...
— sstring_length retourne la taille d'une chaîne,
— sstring_concat retourne la concaténation des deux chaînes.

2.2 read

Cet opérateur lit un chunk sur l'entrée standard et le met sur le haut de la pile. On ne cherche pas à lire un type particulier. Si un opérateur est lu, alors il est empilé dans un value_block.

2.3 Flux de sortie

Il s'agit de créer une valeur value_out_stream renseignant un FILE * et d'ajouter les opérateurs suviants :
— out_stream_open prend un nom de fichier et retourne un value_out_stream correspondant ;
— out_stream_print prend un value_out_stream puis une value et fait un print sur le flux ; et
— out_stream_close prend un value_out_stream et le ferme.

2.4 Localité et empilement de dictionnaires

Il s'agit de chaîner les dictionnaires. Le get cherche dans le premier dictionnaire, s'il ne trouve pas dans le suivant... Le set agit toujours sur le premier dictionnaire (*i.e.* on ne modifie pas la valeur plus loin, on la masque par une nouvelle). On ajoute les opérateurs suivants :
— [ouvre un nouveau dictionnaire (il devient le premier) ;
—] ferme le premier dictionnaire et le supprime (le second devient le premier) ;
— set_up agit comme set mais sur le premier dictionnaire où la variable est définie. Si elle n'est définie nulle part, alors elle l'est sur le dernier dictionnaire (global).

3 Travail demandé (par groupe de trois ou quatre étudiants)

Les fichiers en-tête fournis (.h) sont à respecter *scrupuleusement*.
Il doit être remonté sur Celene un fichier nommé PASD_mini-projet.tzg[4] contenant :
— tous les fichiers sources (.h et .c) ainsi que le Makefile,
— un document compte-rendu.pdf d'au maximum 5 pages, et
— tous les fichiers annexes pour tester d'éventuelles extensions.
Il est possible d'ajouter d'autres fichiers en précisant lesquels et pourquoi dans le compte-rendu.

Les projets seront évalués en fonction du nombre de participants. Pour les groupes de trois étudiants, il n'est pas demandé d'implanter d'extension. Les groupes de quatre étudiants doivent implanter au moins une extension. La formation des groupes est de la responsabilité des étudiants.

Vous pouvez utiliser toutes les bibliothèques standards du C99 (celles qui n'ont pas besoin d'être indiquées à gcc par des -l). Aucune autre bibliothèque ne doit être utilisée.

4. Archive tar -czf qui peut être engendré par make archive.

3.1 Code

Chaque fichier source doit contenir le nom des auteurs en copyright dans les commentaires. Il doit également contenir tous les commentaires nécessaires. Le code doit être lisible.

Le projet est en C99 pur, il doit compiler avec gcc et les arguments -std=c99 -Wall -Wextra... sans aucun *message d'alerte* (et encore moins d'erreur) comme prévu dans le Makefile.

Il ne doit y avoir *aucune constante magique* (*i.e.* seuls 0, 1, et -1 peuvent apparaître dans votre code en dehors des #define).

Il ne doit y avoir *absolument aucune fuite mémoire* (et encore moins de segmentation fault) et toutes les mémoires allouées doivent être rendues avant la fin de l'exécution du programme. Ceci est testé par le makefile.

3.2 Compte-rendu

Il doit comporter les noms des membres de l'équipe, répondre aux questions du sujet, préciser les difficultés rencontrées (et leur solution ou absence de), les éventuelles extensions considérées et les limites de ce qui a été réalisé. Il doit également indiquer ce qui a pu être appris / acquis / découvert durant le projet et le *travail réalisé par chacun*.

3.3 Calendrier

Ce mini-projet est conçu pour être fait rapidement. Il ne faut pas traîner. La restriction du temps fait partie de l'exercice.

Début du projet :	mardi 6 octobre
Remise blanche :	lundi 23 octobre (minuit)
Retour sur remise blanche :	lundi 26 octobre
Remise finale :	**lundi 2 novembre (minuit)**

Il est prévu une *remise blanche* avant la remise finale. Elle n'est pas obligatoire et n'entre pas dans l'évaluation. Elle permet d'avoir un retour succinct pour corriger.

La notation pourra être individuelle. Il n'est pas prévu pour l'instant de soutenance de projet. Mais cela pourra être le cas, en particulier en cas de *zones d'ombre*.

4 Conseils

Cet énoncé et la documentation est un cahier des charges *strict*, toute déviation sera pénalisée, d'autant plus qu'une partie de la correction sera automatisée.

La meilleure organisation est de considérer la remise blanche comme la remise finale. Si la date est tenue, cela permet d'avoir un retour et de finaliser pour la remise finale. Si la date n'est pas tenue, la seule conséquence est que l'on se prive d'un retour, d'une chance d'identifier les problèmes et de les corriger (surtout si ce sont des problèmes comme un mauvais nom de fichier, une archive .tgz corrompue... ou autres « non remises »).

Si vous n'arrivez pas à boucler le projet, rendez le quand même en indiquant (dans le compte-rendu) ce qui a été fait et ce qui ne l'a pas été. De même *si tout ne marche pas parfaitement*, rendez le projet en indiquant ce qui coince.

Il vaut mieux un projet *honnête* qu'un projet *dopé*. Il vaut mieux un début de projet parfait qu'un projet entier qui échoue au premier test. Le but est la maîtrise de la programmation en C, en particulier des pointeurs et de la gestion de la mémoire.

4.1 Pour ceux qui auraient des doutes ou des « tentations »

Dans le cas de l'utilisation d'un autre langage que le C99, ou du non respect du cahier des charges, le mini-projet sera « non remis », *i.e.* noté zéro.

Les « partages » de code entre groupes et les « emprunts » de code sur internet seront à expliquer et justifier dans le compte-rendu (ou à défaut en *Section disciplinaire du conseil d'administration compétente à l'égard des usagers*).