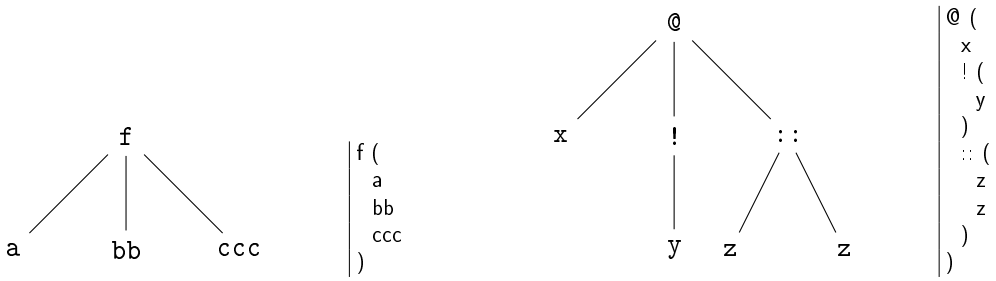
Mini-projet C99 : Manipulation de termes *ver. 0.9*

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 sont disponibles sous Celene.

Le but de ce mini-projet est de manipuler des termes. Un terme est composé d'un symbol éventuellement suivi d'une séquence d'autres termes entouré de parenthèses. Ces sous-termes sont appelés *arguments* et ne sont séparés que par des espaces. En voici trois exemples : t1, f (a bb ccc), et @(x !(y) : :(z z)). Ce sont des structures syntaxiques qui se représentent sous forme d'arbre d'arité non bornée comme sur la Fig. 1.



(a) f(a bb ccc) (b) @(x !(y) : :(z z))

FIGURE 1 – Représentations arborescente et verticale de termes.

Module utilitaire

Les chaînes de caractères sont manipulées sous forme de sstring qui enregistrent taille et tableau. Comme dans le premier TDM, il n'y a pas de zéro final dans le tableau.

make t_sstring et make m_sstring permettent de tester le module string (sortie et mémoire).

1 Termes

Les termes sont représentés par une structure cachée (term_struct) dans le module term. De l'extérieur, on ne dispose que de pointeurs sur ce type structure : term. Un term_struct se compose d'un symbole (sstring), du nombre d'arguments (arity), d'un lien vers le terme père (si c'est un attribut) et de liens vers les extrémités d'une liste doublement chaînée de termes (term_list) qui enregistrent les arguments.

Le fichier test_term.c contient le code pour créer les trois termes qui servent d'exemple plus haut.

Les termes sont également pourvu d'un mécanisme permettant de parcourir leurs arguments de gauche à droite avec les term_argument_traversal (en suivant le patron des *itérateurs*).

(1) Dans le module term, les pré-conditions se contentent d'exiger qu'aucun terme ne soit NULL alors que l'on pourrait tester sa validité complète (c.-à-d. de tout le terme). Sans écrire la fonction, expliquer comment fonctionnerait une fonction servant à cela. Donner sa complexité en fonction de a le nombre de symboles dans le terme et l le nombre moyen de lettres d'un symbole.

1.1 Entrées et sorties

Le module term_io contient les fonctions d'entrée (term_scan) et de sortie (term_print_...). Il y a deux formats de sortie, compact (sur une ligne) et vertical (voir Fig. 1).

Il faut écrire ce module pour pouvoir commencer à utiliser test_term.

make t_term et make m_term permettent de tester les modules term et term_io (sortie et mémoire).

1.2 Variables

Certains symboles sont distingués et jouent le rôle de variables. Ils doivent s'écrire 'xxx où xxx est un identifiant C valide. Si le symbole est identifié comme une variable alors l'arité du terme doit être de zéro. Le module term_variable permet de tester si un terme est une variable et offre la possibilité de remplacer partout une variable par sa valeur (toutes les valeurs sont des termes).

make t_variable et make m_variable permettent de tester le module term_variable (sortie et mémoire).

1.3 Donner des valeurs à des variables (valuate)

Il s'agit de traiter tous les sous-termes de la forme : set (<*variable*> <*valeur*> t). Ceux-ci entraîne remplacement de <*variable*> par <*valeur*> chaque fois qu'elle apparaît dans t.

Si la même variable est redéfinie par un set dans un sous-terme, cela entraîne le remplacement par la nouvelle valeur pour le sous-terme. Pour gérer cela, les variables à remplacer sont empilées : une nouvelle déclaration masque alors les plus anciennes.

Attention dans la valuation, il faut traiter le cas où un remplacement introduit de nouvelles variables à remplacer ou de nouvelles définitions. Ces variables ne sont pas remplacées quand on découvre le set mais quand on fait le remplacement de la variable.

Pour éviter les remplacements à l'infini, on doit s'assurer que <*variable*> n'apparaît pas dans <*valeur* > (donc pas de set ('a R('a) BB)).

1.4 Unification (unify)

Il s'agit de résoudre un système d'égalité entre termes contenant des variables ou d'indiquer pourquoi il n'est pas résolvable. Le principe est de prendre les égalités entre termes en séquence.

Si l'un des deux cotés est une variable alors, soit :

- elle est aussi égale à l'autre coté (p.e. =('a 'a)), c'est trivialement vrai, on passe à la suite,
- elle apparaît aussi de l'autre coté (mais plus bas comme dans =('a f ('a))) alors il ne peut y avoir de solution, on rejette en indiquant cette égalité,
- elle n'apparaît pas de l'autre coté, alors on sort cette égalité comme valeur pour la variable et on la remplace partout (égalités restant à traiter et valeurs pour d'autres variables déjà trouvées) par cette valeur,

Si aucun des deux cotés n'est une variable. Si les symboles ou les arités sont différents, on rejette en indiquant cette égalité. Sinon, on ajoute à la fin de la séquence des égalités les égalités entre les premiers termes, les seconds termes...

(2) Pourquoi cet algorithme s'arrête-il ? Par quoi peut-on borner le nombre d'égalités que l'on va devoir traiter ?

(3) Quelle peut être l'utilité d'un tel algorithme ?

1.5 Ré-écriture de terme (rewrite)

Le terme présenté à rewrite doit être de la forme :

rewrite (-> (<*a_trouver*> <*remplacer_par*>) t)

Il s'agit de remplacer n'importe quelle occurrence du terme <*a_trouver*> par un terme <*remplacer_par*> dans le term t. Le choix de l'occurrence remplacée est non déterministe. Le résultat n'est donc pas un terme mais l'ensemble de tous les résultats possibles.

Par exemple, pour rewrite (-> (d(t) AA) || (d(t) d(u) d(t))) on recupère
results (|| (AA d(u) d(t) || (d(t) d(u) AA))) car d(t) se trouve à deux endroits.

Il peut y avoir plusieurs règles, chacune a un symbol -> avant t. Dans le résultat se trouve les résultats obtenus en remplaçant une fois une occurrence d'une règle. Par exemple, pour
rewrite (->(d(t) AA) ->(d(u) BB)]](d(t) d(u) d(t))) on recupère
results ( ]](AA d (u) d (t)]](d (t) BB d (t)))]](d (t) d (u) AA)))
car il y a deux endroits où l'on peut remplacer d(t) et un pour d(u).

Les règles peuvent comporter des variables. Elles prennent leur valeur quand on identifie le motif. Si une variable apparaît plusieurs fois dans le motif, elle doit toujours y avoir la même valeur. On remplace dans <*remplacer_par*> toutes les variables valuées par l'identification du motif.

Par exemple, pour rewrite (-> (d ('a) W ('a 'b))][(d (!) d (@))) on recupère
results ( ][(W (! 'b) d (@))][(d (!) W (@ 'b))) dans le premier 'a vaut ! et @ dans le second.

Enfin les règles de ré-écriture peuvent être précédées par un entier. C'est le nombre d'application de règles (en tout et non par règle).

Attention, les arguments de results sont triés par ordre croissant et il n'y a pas deux termes identiques.

1.6 Makefile pour les derniers modules

Dans le répertoire DATA se trouvent un répertoire contenant des termes (Terms) et un autre contenant les sorties attendues (Results_Expected).

Sans argument, make affiche un résumé de ce qu'il propose :
— TVn teste la sortie de test_valuate sur l'entrée t_valuate_n.term,
— MVn teste la gestion de la mémoire par test_valuate sur l'entrée t_valuate_n.term,
— TV teste les sorties de test_valuate sur chacune des entrées t_valuate_*.term,
— MV teste la gestion de la mémoire par test_valuate sur chacune des entrées t_valuate_*.term,
— V fait TV et MV,
— de même TUn, MUn, TU, MU et U pour unify,
— de même TRn, MRn, TR, MR et R pour rewrite,
— T fait tous les tests sur les sorties, et
— M fait tous les tests de mémoire.

2 Extensions

Il n'y a pas de programmes pour tester les extensions ni même de fichiers entêtes. Ceux-ci doivent donc être fournis (avec toute la documentation et les commentaires nécessaires), ainsi que les fichiers annexes pour les faire fonctionner et les tester (modifier Makefile pour assurer l'inclusion de ces fichiers).

2.1 Productions par ré-écriture

Il s'agit de reprendre le système de ré-écriture et de permettre l'ajout de * avant les règles. Le résultat est l'ensemble des termes que l'on a pu écrire par un nombre quelconque de ré-écritures.

Il faut aussi permettre l'ajout de ** avant les règles. Le résultat est l'ensemble des termes que l'on a pu écrire par un nombre quelconque de ré-écritures et pour lesquels aucune nouvelle ré-écriture n'est possible.

Enfin on donnera la possibilité de faire suivre un entier naturel qui est le nombre maximal de ré-écriture à considérer.

(4) Donner des exemples où les résultats renvoyés par * et ** ne seraient pas des ensembles finis.

(5) Est-il possible de prévoir si le nombre de ré-écriture sera fini ou non ?

2.2 Expressions

Il s'agit de restreindre les symboles à +, -. . . !, &&. . . ainsi qu'aux entiers et aux booléens et de pouvoir évaluer un terme pour obtenir un entier ou un booléen.

(6) Sans ajouter d'autres types, proposer (sans les implanter) d'autres symboles et leur sémantique pour aboutir à un langage de programmation (et non simplement de calcul).

2.3 Arithmétique de Peano

Il s'agit de coder les entiers naturels par 0 et S. Le nombre 3 se représente par S(S(S(0))) ; le symbole S veut dire *plus un*.

Il faut implanter l'addition et la multiplication sous ce format avec des règles de ré-écriture. Puis ajouter des constructions du type « Si c'est zéro A sinon B. »

(7) Proposer (sans implanter) des symboles et leur sémantique pour implanter les fonctions récursives (de la théorie de la récursion).

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

Il faut écrire tous les modules pour les termes (jusqu'à rewrite). L'ordre de présentation de cet énoncé correspond aux dépendances logiques et à la difficulté croissante. En particulier les derniers modules demandent une bonne compréhension des objets manipulés, des algorithmiques à implanter et des fonctions déjà développées (ou proposées au développement) avant d'allumer un ordinateur.

Il doit être remonté sur Celene un fichier nommé PASD_mini-projet.tzg[1] contenant :
— tous les fichiers sources (.h et .c) ainsi que le Makefile et
— un document compte-rendu.pdf d'au maximum 5 pages.

Il est possible d'ajouter d'autres fichiers en précisant lesquels et pourquoi dans le compte-rendu.

Les projets sont 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.

1. Archive tar -czf qui peut être engendrée par make archive.

3.1 Code

Les fichiers en-tête fournis (.h) sont à respecter *scrupuleusement*.

Chaque fichier source doit contenir tous les commentaires nécessaires. Le code doit être lisible. Si des fichiers sont ajoutés, ils doivent contenir toute la documentation nécessaire au format Doxygene.

Le projet est en C99 pur. Seules les bibliothèques standards du C sont autorisées. La projet 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 fuite mémoire* (et encore moins de segmentation fault) et toutes les mémoires allouées doivent être rendues avant la fin de l'exécution du programme.

3.2 Compte-rendu

Il doit comporter les noms des membres du groupe, 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 car il demande un investissement conséquent. La restriction du temps fait partie de l'exercice.

Début du projet :	mardi 11 octobre
Remise blanche :	vendredi 28 octobre (minuit)
Retour sur remise blanche :	mercredi 2 novembre
Remise finale :	**lundi 7 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 [2].

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 ».

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

2. Surtout si ce sont des problèmes comme un mauvais nom de fichier, une archive .tgz corrompue... ou autres « non remises ».