Correction du partiel CMP2

EPITA – Apprentis promotion 2011
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices et ordinateurs interdits.

Juillet 2009 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante. Une lecture préalable du sujet est recommandée.

Écrivez votre nom en haut de la première page du sujet, et rendez-le avec votre copie.

1 Généralités

1. Qu'est-ce que le sucre syntaxique ?

   Correction: Il s'agit de constructions syntaxiques supplémentaires proposées par un langage n'apportant rien de plus au niveau de sa syntaxe abstraite, mais permettant des variantes d'écriture. Le sucre syntaxique apporte souvent simplicité, lisibilité, raccourcis d'écriture, confort, etc. Il peut permettre l'expression de nouveaux concepts (la surcharge de fonctions peut être considérée comme telle, par exemple).

2. En donner un exemple (dans le langage de votre choix).

   Correction:
   – Les opérateurs '+=', '−=', etc. en C.
   – En Tiger, '**if** a **then** b', qui se désucre en '**if** a **then** b **else** ()'
   – Les générateurs d'Haskell : [(x,y) | x <− [1 .. 6], y <− [1 .. x], x+y < 10]

3. Qu'est-ce que le sel syntaxique ?

   Correction: Le dual du sucre syntaxique : des constructions relevant purement de la syntaxe concrète et n'ajoutant rien à l'expressivité du langage et qui servent généralement de garde-fou pour permettre au compilateur d'aider son utilisateur en cas d'erreur.
   Notez qu'une reprise sur erreur efficace dans un parseur engendré par Bison s'appuie généralement sur des constructions de sel syntaxique. Par exemple si une erreur se produit en entre '(' et ')', il peut être judicieux d'ignorer tous les tokens entre ces deux symboles.

4. En donner un exemple (dans le langage de votre choix).

   Correction: Par exemple, les éléments de ponctuation (virgules, points-virgules, deux-points, etc.) servant de séparateur ou de terminateur. Ou encore les instructions de fin de bloc du Bourne Shell (**fi**, **done**, etc.).

5. À quoi sert `make` ? (Note : se contenter de répondre « à compiler » ne rapportera pas de point.)

   Correction: Je cite la page `man` de GNU Make :

   > The purpose of the `make` utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. (...) Our examples show C programs, since they are most common, but you can use `make` with any programming language whose compiler can be run with a shell command. In fact, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

   En résumé, `make` est un outil de gestion de projet, au sens large. Il permet de décrire (via un fichier `Makefile`) comment produire/construire certains fichiers (appelés *cibles*) grâce à des *règles*. Une règle est composée d'une liste de *dépendances* (des cibles qui doivent êtres construite avant et être à jour vis-à-vis de la cible initiale) et d'une ou plusieurs *action(s)* (commandes shell qui ont pour objet de construire effectivement la cible).
   Grâce à ce mécanisme purement déclaratif, `make` sait ce qu'il est nécessaire de construire lorsqu'il est appelé avec une cible donnée, en établissant un graphe de dépendances entre les différentes cibles. À retenir : à la différence d'un shell script, `make` n'a pas pour objet d'exécuter des commandes dans un ordre imposé par le programmeur. Au contraire, l'objet de `make` est d'effectuer le plus petit nombre d'actions pour construire une cible donnée. C'est notamment pour cela qu'on l'utilise dans des projets logiciels pour faire de la compilation séparée/incrémentale.

6. À quoi peut bien servir le bout de code C++ ci-dessous ?

```
template <typename T>
struct fun
{
 typedef T ret;
};

template <typename T>
struct fun<T*>
{
 typedef typename fun<T>::ret ret;
};
```

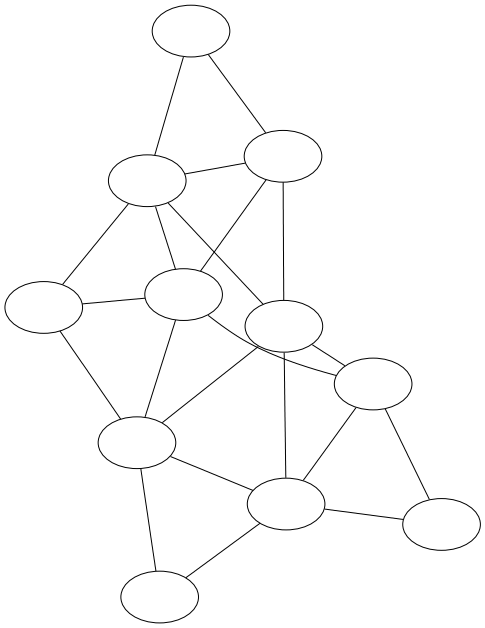
   (Donner un exemple d'utilisation est une bonne idée.)

   Correction: Il s'agit d'un *traits* C++ (une sorte de fonction agissant sur des types) permettant de retrouver le type de base correspondant à type pointeur donné. Dit autrement, `fun` "retire les étoiles d'un type".
   Ce traits s'utilise comme ceci :

```
typename fun<float*>::ret bar;   // 'bar' is 'float'.
typename fun<int***>::ret baz;   // 'baz' is 'int'.
typename fun<double>::ret quux;  // 'quux' is 'double'.
```

   Notez que cela fonctionne aussi avec les types non pointeurs (cf. le dernier cas). Ce genre de constructions peut-être utile pour obtenir (statiquement) de l'information sur un type `T` donné, dont peut tirer parti un code C++.

2 Allocation de registres



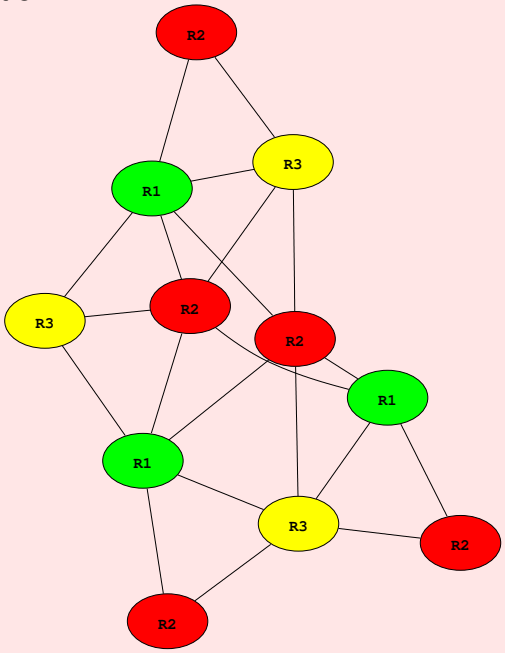
Colorer ce graphe d'exclusion mutuelle en 3 registres : R1, R2, R3.

Rendre le sujet en ayant annoté chaque nœud d'un R1, R2, R3.

L'utilisation de stylos de couleurs différentes n'est pas requise, mais sera appréciée.

N'oubliez pas d'écrire votre nom en haut de la première page.

Correction:



3 Élimination de fonctions inutilisées

Une opération simple qu'un compilateur peut effectuer sur un programme est l'élimination de fonctions inutilisées. Par exemple, dans le programme C++ suivant, `bar` n'est pas utilisée et on pourrait l'éliminer sans changer le comportement du programme.

```
static void foo () {}
static void bar () {}
int main () { foo(); }
```

1. Citer les différentes phases de la partie frontale (*front end*) d'un compilateur comme `tc`.

   Correction: (i) scanner, (ii) parser et construction de l'Abstract Syntax Tree (AST), (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

2. Même question avec la partie terminale (*back end*).

   Correction: (i) sélection d'instruction (génération de code), (ii) analyse de vivacité, (iii) allocation de registres.
   NB : Il est plus correct de considérer que la canonisation de la représentation intermédiaire fait partie d'une tierce partie, dite centrale (*middle end*).

3. À quel(s) endroit(s) dans le compilateur pourrait-on place placer une étape d'élimination de fonctions inutilisées ?

   Correction: Normalement, juste après la phase de liaison (*binding*) : la seule information nécessaire pour déterminer la non utilisation d'une fonction est l'absence de son *nom* dans un appel de fonction.
   Dans le cas d'un langage présentant la surcharge de fonctions (par exemple Tiger lorsque l'option '`-overfun-types-compute`' de `tc` est activée), il faut un peu plus d'information : une fonction est alors identifiée par son nombre d'arguments et leurs types, en sus de son nom. Il est donc nécessaire d'attendre la phase de vérification des types pour éliminer des fonctions surchargées.

4. Quel(s) avantage(s) y a-t-il à effectuer une telle opération ? (Il y a au moins deux bonnes réponses, mais une seule suffit.)

   Correction:

   Accélérer le compilateur En retirant des (définitions de) fonctions, on élague l'AST, ce qui signifie moins de données à parcourir et à traiter dans les phases en aval de cet étape (les fonctions éliminées ne sont plus "regardées" par le compilateur). En particulier, il sera inutile de vérifier les types des nœuds du sous-arbre correspondant à cette fonction, de traduire ce sous-AST en représentation intermédiaire, etc.

   Réduire la taille du programme produit Puisque le code des fonctions éliminées n'est plus traité après l'élagage, celles-ci ne donnent plus lieu à génération de code. La taille du programme généré s'en trouve donc réduite. Une telle opération est parfois utile/nécessaire dans certains contextes (pour des contraintes matérielles, afin d'optimiser les performances du cache, etc.).

5. Dans quel cadre de travail spécifique on ne voudrait surtout pas utiliser une telle optimisation ? (Il n'est pas impossible qu'un indice soit présent dans ce sujet.)

   Correction: Par exemple si l'on désire réaliser une bibliothèque de fonctions (C++ par ex.) Bon nombre des fonctions ne seront probablement jamais appelées par la bilbiothèque, et l'on veut pourtant les fournir à l'utilisateur !
   C'est d'ailleurs pour cette raison que le code donné au début de l'exercice fait apparaître le terme '`static`' : ces fonctions ne sont pas destinées à être exportées, et libre au compilateur de les ignorer si elles ne sont pas utilisées dans l'unité de compilation courante (le fichier objet).
   NB : Ce problème ne se pose pas en Tiger, puisque le langage et le compilateur ne supportent pas la compilation séparée ; s'il y a une éventuelle phase d'édition de lien (dans le back end IA-32), il s'agit purement d'un détail d'implémentation, et non d'une fonctionnalité.

6. Si l'on voulait implémenter l'élimination de fonctions inutilisées dans `tc`, quels seraient les types de nœuds de l'AST à considérer ?

Correction: Sans trop de surprise, l'implémentation de l'élagueur va reposer sur un visiteur d'AST. Les nœuds à considérer sont :

`CallExp` L'appel d'une fonction est l'information qui permet de savoir qu'une fonction est effectivement utilisée. Chaque fonction appelée doit être retenue ou étiquetée comme telle.

`FunctionDec` (et `MethodDec`) Tous les visiteurs de `tc` qui opèrent une transformation travaillent par copie : l'AST original n'est jamais modifié. Pour pouvoir éliminer une fonction inutilisée, il est nécessaire de faire le lien entre le nœud `FunctionDec` initial (parcouru par le visiteur) et le nouveau nœud `FunctionDec` cloné (qui sera éventuellement supprimé plus tard).

`LetExp` L'essentiel du travail se situe ici. La portée d'une fonction est délimitée par le bloc `let-in-end` qui l'englobe, et c'est donc à partir de `LetExp` que va s'effectuer la recherche de fonctions inutilisées. Comme rappelé précédemment, le visiteur procède par copie de l'AST initial. Puis toutes les déclarations de fonctions (entre `let` et `in`) clonées sont inspectées et supprimées si marquées comme inutilisées.

7. Comment pourrait-on détecter qu'une fonction peut être éliminée dans `tc` ?

   Correction: Il en a déjà été un peu fait mention dans la correction de la question précédente : en s'assurant qu'elle n'est jamais appelée, autrement dit qu'il n'existe aucun nœud `CallExp` dont le site de définition est précisément la `FunctionDec` (ou `MethodDec`) associée à cette fonction.
   Exceptions notables : `_main` (qui n'est "visiblement" appelée par personne, mais qu'il faut bien conserver) ; et les builtins, qu'on ne peut pas vraiment retirer puisque leur code n'est pas écrit par l'utilisateur.

8. Surprise (à moitié) ! Les plus attentifs auront remarqué que cette option est *déjà* présente dans le compilateur Tiger de référence, et vous pouviez l'implémenter dans le cadre d'une extension optionelle du projet. Dans le sujet, ce travail facultatif figurait dans la section « TC-I, Function inlining ». Rappelez en quoi consiste l'optimisation de mise en ligne du corps des fonctions (*inlining of function bodies*).

   Correction: (Pour information, ce visiteur "élagueur" s'appelle `Pruner` dans `tc`.)
   La *mise en ligne* du corps d'une fonction (*inlining*) est une optimisation consistant à remplacer un appel de fonction ou de méthode par le corps de celle-ci au site d'appel (en remplaçant les arguments formels par les arguments effectifs).
   En procédant ainsi, on supprime les opérations inhérentes à un appel de fonction (prologue/épilogue) : copie des arguments sur la pile, mise à jour des pointeurs `fp` et `sp`, sauvegarde éventuelle de certains registres (sous la responsabilité de l'appellant, dits *callee-save*), etc. Il s'agit donc d'un gain de vitesse.
   De plus, la mise en ligne permet des optimisation *interprocédurales* dans un modèle de compilation où les fonctions sont compilées séparément. Normalement, le compilateur ne devrait jamais disposer en même temps des informations de la fonction appelante et de la fonction appelée. En mettant en ligne la seconde au sein de la première, on casse cette barrière entre fonctions, et on peut aider le compilateur à effectuer des optimisation autrement impossible à réaliser : propagation de constantes, élimination de code mort, etc.

9. À votre avis, pourquoi l'élimination de fonctions inutilisées est-elle située dans la même section que l'inlining ?

   Correction: Parce que notre version de la mise en ligne ne supprime pas les fonctions effectivement mises en ligne ; il était dommage de les conserver après leur mise en ligne (cf. question), d'où cette passe d'élagage post-inlining.

10. Dans quel(s) cas est-il impossible de mettre en ligne une fonction ?

    Correction:
    – La définition de la fonction n'est pas visible depuis le site d'appel (c'est-à-dire, dans la même unité de compilation).
    – La fonction candidate à la mise en ligne est récursive (directement ou indirectement) : la mise en ligne ne peut avoir lieu, car il s'agirait d'un processus sans fin. (Bien entendu, on pourrait proposer de limiter la profondeur d'inlining à un certain nombre d'appels récursifs, mais ça commence à devenir trop subtil pour une question qui se voulait initialement simple !)
    – On ne peut mettre en ligne une méthode polymorphe (c'est-à-dire une fonction membre virtuelle en C++), car sa liaison est effectuée à l'exécution, alors que la mise en ligne est une opération effectuée à la compilation.
    – Un peu pour les mêmes raisons, on ne peut mettre en ligne une fonction appelée via un pointeur sur fonction.
    – Enfin, il se trouve que certains compilateurs refusent de mettre en ligne une fonction dès lors qu'elle accepte un nombre variable d'arguments (opérateur « `...` » du C++). Sans être impossible, ce choix d'implémentation est compréhensible car le passage et la récupération d'arguments variables s'effectuent via une manipulation du cadre de pile, dont la traduction lors de la mise en ligne pourrait être non triviale.

11. Il est possible de supprimer encore plus de code inutilisé. Considérez l'exemple ci-dessous.

```
static void foo () {}
static void bar () { foo(); }
int main () {}
```

    Ici, `foo` est bien utilisée par `bar`, mais cette dernière n'est pas utilisée. On pourrait donc supprimer ces deux fonctions et obtenir un programme équivalent après compilation. Proposez un algorithme pour généraliser cette approche.

    Correction: Seule l'idée d'un tel algorithme est suggérée ici : celle-ci consisterait à supprimer toutes les fonctions appartenant aux composantes connexes du graphe d'appel de fonctions (*call graph*) qui ne contiennent ni définition de `_main` ni une définition de *builtin*.

4 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire. Les renseignements ci-dessous ne seront bien entendu pas utilisés pour noter votre copie. Ils ne sont pas anonymes, car nous souhaitons pouvoir confronter réponses et notes. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Sauf indication contraire, vous pouvez cocher plusieurs réponses par question. Répondez sur la feuille de QCM qui vous est remise. N'y passez pas plus de dix minutes.

Le cours

1. Quelle a été votre implication dans les cours CMP1, CMP2 et TYLA ?

   A Rien.

   B Bachotage récent.

   C Relu les notes entre chaque cours.

   D Fait les annales.

   E Lu d'autres sources.

2. Ce cours
   A Est incompréhensible et j'ai rapidement abandonné.
   B Est difficile à suivre mais j'essaie.
   C Est facile à suivre une fois qu'on a compris le truc.
   D Est trop élémentaire.
3. Ce cours
   A Ne m'a donné aucune satisfaction.
   B N'a aucun intérêt dans ma formation.
   C Est une agréable curiosité.
   D Est nécessaire mais pas intéressant.
   E Je le recommande.
4. La charge générale du cours en sus de la présence en amphi (relecture de notes, compréhension, recherches supplémentaires, etc.) est
   A Telle que je n'ai pas pu suivre du tout.
   B Lourde (plusieurs heures par semaine).
   C Supportable (environ une heure de travail par semaine).
   D Légère (quelques minutes par semaine).

Les formateurs

5. L'enseignant
   A N'est pas pédagogue.
   B Parle à des étudiants qui sont au dessus de mon niveau.
   C Me parle.
   D Se répète vraiment trop.
   E Se contente de trop simple et devrait pousser le niveau vers le haut.
6. Les assistants
   A Ne sont pas pédagogues.
   B Parlent à des étudiants qui sont au dessus de mon niveau.
   C M'ont aidé à avancer dans le projet.
   D Ont résolu certains de mes gros problèmes, mais ne m'ont pas expliqué comment ils avaient fait.
   E Pourraient viser plus haut et enseigner des notions supplémentaires.

Le projet Tiger

7. Vous avez contribué au développement du compilateur de votre groupe (une seule réponse attendue) :
   A Presque jamais.
   B Moins que les autres.
   C Équitablement avec vos pairs.
   D Plus que les autres.
   E Pratiquement seul.
8. La charge générale du projet Tiger est

A Telle que je n'ai pas pu suivre du tout.
B Lourde (plusieurs jours de travail par semaine).
C Supportable (plusieurs heures de travail par semaine).
D Légère (une ou deux heures par semaine).
E J'ai été dispensé du projet.

9. Y a-t-il de la triche dans le projet Tiger ? (Une seule réponse attendue.)
   A Pas à votre connaissance.
   B Vous connaissez un ou deux groupes concernés.
   C Quelques groupes.
   D Dans la plupart des groupes.
   E Dans tous les groupes.

Questions 10-16 Le projet Tiger vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille qui suit. (Une seule réponse attendue par question.)

A Pas du tout
B Trop peu
C Correctement
D Bien
E Très bien

10. Formation au C++.
11. Formation à la modélisation orientée objet et aux *design patterns*.
12. Formation à l'anglais technique.
13. Formation à la compréhension du fonctionnement des ordinateurs.
14. Formation à la compréhension du fonctionnement des langages de programmation.
15. Formation au travail collaboratif.
16. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 17-29 Comment furent les étapes du projet ? Répondre selon la grille suivante. (Une seule réponse attendue par question ; ne pas répondre pour les étapes que vous n'avez pas faites.)

A Trop facile.
B Facile.
C Nickel.
D Difficile.
E Trop difficile.

17. Rush `.tig` : mini-projet en Tiger (Bistromatig).
18. `TC-0`, Scanner & Parser.
19. `TC-1`, Scanner & Parser, Tâches, Autotools.
20. `TC-2`, Construction de l'AST et pretty-printer.
21. `TC-3`, Liaison des noms et renommage.
22. `TC-E`, Calcul des échappements.
23. `TC-4`, Typage.

24. Désucrage des constructions objets (transformation Tiger → Panther).
25. `TC-5`, Traduction vers représentation intermédiaire.
26. Option `TC-A`, Surcharge des fonctions.
27. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
28. Option `TC-B`, Vérification dynamique des bornes de tableaux.
29. Option `TC-I`, Mise en ligne du corps des fonctions.