Votre nom :

Correction du Partiel TYPO & CMP
Typologie des Langages
Construction des Compilateurs

EPITA – Promo 2007 – **Tous documents autorisés** *

Juin 2005 (1h30)

Le sujet et une partie de sa correction ont été écrits par Akim Demaille.

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. Rendez ce sujet avec votre nom dûment mentionné ci-dessus.

1 Typologie des Langages

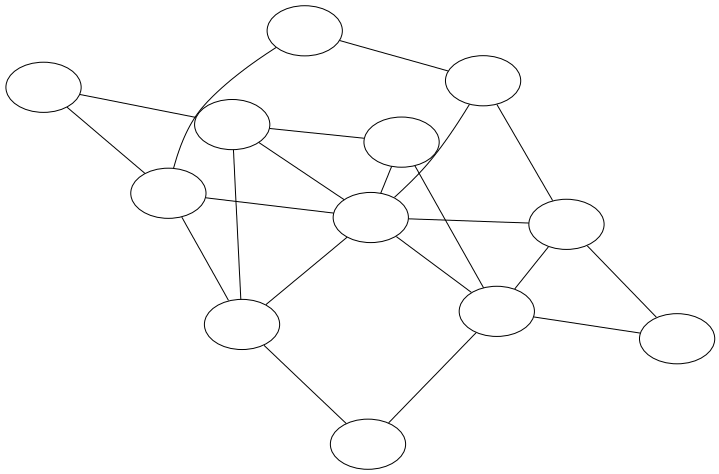
1. Quelles différences y a-t-il entre un `boost::variant` et une simple `union` en C++ ?
2. Quel *design pattern* est parfaitement adapté pour l'écriture de fonction sur des variants en C++ ? Expliquer.
3. Discuter les avantages relatifs du passage d'argument par valeur-résultat d'une part, et par référence d'autre part.
4. Dans le cas du C++, comment tirer le meilleur parti de chacun (valeur-résultat/référence) dans une fonction générique ?
5. En Eiffel, lorsqu'une méthode est redéfinie (i.e., l'équivalent de `virtual` en C++), on a droit à :

   (a) `require then`
   (b) `require else`
   (c) `ensure then`
   (d) `ensure else`

   Correction: Si une méthode prétend se substituer à une autre, il lui faut accepter toutes les acceptations de son prédécesseur. Par conséquent les préconditions peuvent être affaiblies : `require else`. Pour les mêmes raisons, les postconditions peuvent être plus strictes : `ensure then`.

*"Tout document autorisé" signifie que notes de cours, livres, annales, etc. sont explicitement consultables pendant l'épreuve. Le zèle de la part des surveillants n'a pas lieu d'être, mais dans un tel cas contacter le LRDE au 01 53 14 59 22.

2 Construction des Compilateurs : Allocation des Registres

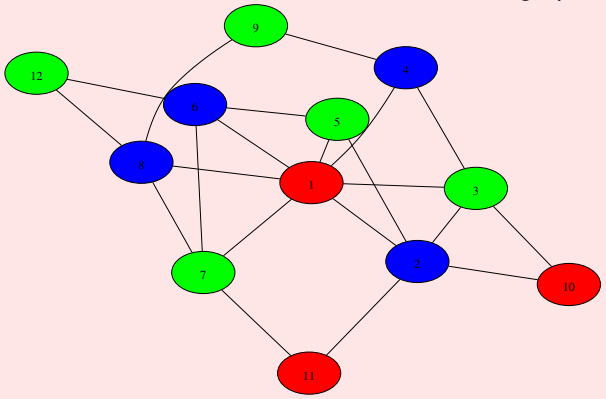


Colorer ce graphe en 3 registres : rouge, vert et bleu.

En guise de réponse, rendre le sujet en ayant annoté chaque noeud d'un V, R, ou B selon qu'il est vert, rouge ou bleu (ou violet, rose et bigarré selon vos goûts).

Écrire votre nom en haut.

Correction: Pour information un ordre de reconstruction du graphe est donné.



3 Construction des Compilateurs : Support de `continue`

L'objet de ce problème est de considérer l'ajout d'une fonctionalité au langage Tiger : `continue`, à l'instar de l'instruction homonyme dans le C et bien d'autres langages.

Les questions suivantes sont posées dans l'ordre des stades de compilation. Notez cependant que certains modules tardifs nécessitent une collaboration de modules plus en amont : il est plus sain de réfléchir globalement à toutes les questions, puis seulement de répondre dans l'ordre.

Si une étape ne nécessite aucune modification pour supporter l'introduction de `continue`, simplement le dire, et ne pas tomber verbeusement dans le piège tendu par la question.

Sémantique Si `break` est bien égal à lui-même dans un `for` et dans un `while`, quelle est la subtile différence pour le cas du `continue` ?

TC-0 : Interface On souhaite activer cette extension par l'option `--continue`. Décrire comment modifier le compilateur pour activer/désactiver le support de `continue` en un minimum d'effort.

> **Correction:** Si `continue` n'est pas supporté, alors quand le scanner le lit, il lui faut déclarer un identificateur, un symbole. Donc dans le scanner :
>
> ```
> "continue"   {
>   if (enable_continue_support)
>     return TOKEN_CONTINUE;
>   else
>     {
>       yylval->symbol = &symbol::Symbol::create (yytext);
>       return TOKEN_IDENTIFIER;
>     }
> }
> ```
>
> et c'est tout !

TC-1 : Grammaire Décrire comment modifier la grammaire de Tiger pour accepter `continue`.

> **Correction:** Exactement comme pour le cas de `break`, on ne cherche pas à contraindre dès maintenant l'apparition de `continue` exclusivement dans une boucle.
>
> ```
> exp: continue;
> ```

TC-2 : AST Décrire comment intégrer cette instruction dans l'AST de Tiger.

> **Correction:** Introduire `ContinueExp` dérivant de `Exp`, sans argument, mais avec un attribut (mettons `def_`) pour pointer vers le `ForExp`/`WhileExp` et ses accesseurs.

TC-3 : Liaison Comment modifier le `BindVisitor` pour supporter `continue` ?

> **Correction:** Au même titre que pour le `break`, il faut lier les `continue` à leur `ForExp`/`WhileExp` hôte, de façon à ce que l'on puisse le traduire lors de TC-5 en un `Jump`.
> Il suffit donc d'ajouter une méthode pour visiter les `ContinueExp` qui définisse le `def_` du `continue`. Bien sûr émettre un message d'erreur si jamais on n'a pas de boucle environnante. Le cas des `FunctionDec` qui masquent les boucles est déjà pris en compte pour les `break`.

TC-4 : Typage Décrire les modifications à apporter au `TypeVisittor`.

> **Correction:** Les règles sont les mêmes que pour `break` : le type retourné est `Void`.
>
> ```
> void
> TypeVisitor::operator() (ast::ContinueExp& e)
> {
>   e.type_set (&Void::instance ());
> }
> ```

TC-D : Désucrage des boucles `for` On rappelle que dans le partiel précédant on se proposait de désucrer la boucle `for`

```
for i := l to u do b
```

ainsi :

```
let
  var _l := l'  /* l désucré. */
  var _u := u'  /* u désucré. */
  var i := _l
in
  if i <= _u then
    while 1 do
      (
        b';     /* b désucré. */
        if i = _u then
          break;
        i := i + 1
      )
end
```

Que faire pour la gestion du `continue` ?

TC-5 : Langage intermédiaire Quelle modification apporter au langage intermédiaire `Tree` pour supporter `continue` ?

> **Correction:** On n'a besoin de rien de nouveau.

TC-5′ : Code intermédiaire Quel code produire pour un `continue` ?

> **Correction:** Gros piège ici ! La gestion du `continue` dépend de la nature de la boucle à l'intérieur de laquelle il se trouve.
> Bien sûr, à l'instar du `break`, il faut faire un `jump`, mais à la différence du `break` il faut faire un branchement vers la *tête* de la boucle : il nous faut donc une étiquette nouvelle, `loop_begin`. Ainsi le `continue` devient-il `jump l123` en imaginant que `loop_begin` vaille `l123`.

TC-5″ : Génération du code intermédiaire Comment modifier la génération de code intermédiaire pour le faire ?

> **Correction:** Pour la génération du code, il faut modifier la gestion des boucles pour introduire la gestion de cette étiquette nouvelle. La traduction du `continue` est immédiate.

TC-6 : Canonisation Comment modifier la traduction HIR vers LIR ?

> **Correction:** Pas de changement.

TC-7 : Sélection des Instructions Comment modifier la traduction LIR vers assembleur MIPS ?

> **Correction:** Pas de changement.

TC-8 : Graphe d'Interférence Comment modifier la génération des graphes de flot de contrôles, de vivacité, d'exclusion mutuelle ?

> **Correction:** Pas de changement.

TC-9 : Allocation des Registres Comment modifier l'allocation des registres ?

> **Correction:** Pas de changement.

TC-+ Dans le cas de boucles imbriqués, on voudrait disposer d'un `break`/`continue` plus puissant. Quelles sont vos recommandations ?

> **Correction:** On peut soit avoir un entier pour dire de combien de boucles on sort, ou bien nommer les boucles. La deuxième solution est plus élégante, mais demande plus d'effort d'implémentation. Tout ce que l'on a dit continue de fonctionner normalement, à condition d'accepter d'utiliser une `map` pour stocker les étiquettes des boucles. Cette `map` est soit étiquetée par des entiers (`break 3`) soit par une étiquette (`break outer`).