Votre nom :

Correction du Partiel CMP2
Construction des Compilateurs

EPITA – Promo 2009 – **Tous documents autorisés**

Juin 2007

Le sujet et sa correction ont été écrits par Roland Levillain et Akim Demaille.

Ce partiel se compose de deux parties : la première est l'épreuve de CMP2 elle-même ; la seconde, une évaluation des cours de THL, CCMP et TYLA. Nous vous invitons à répondre à cette dernière, mais n'y consacrez pas trop de temps ; concentrez-vous sur l'épreuve elle-même.

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.

Première partie
Épreuve de Construction des Compilateurs

1 Incontournables

Il n'est pas admissible d'échouer sur une des questions suivantes : **chacune induit une pénalité sur la note finale**.

1. Un sous-langage d'un langage rationnel (i.e., un sous-ensemble) est rationnel. vrai/faux ?

   Correction: N'importe quel langage L sur un alphabet Σ vérifie $L \subset \Sigma^\star$, donc bien sûr que c'est faux.

2. `sizeof` est une fonction de la bibliothèque standard du langage C. vrai/faux ?

   Correction: `sizeof` est un *opérateur* du C, donc c'est faux. Ce ne peut en aucun cas être une fonction, car son application est évaluée à la compilation.

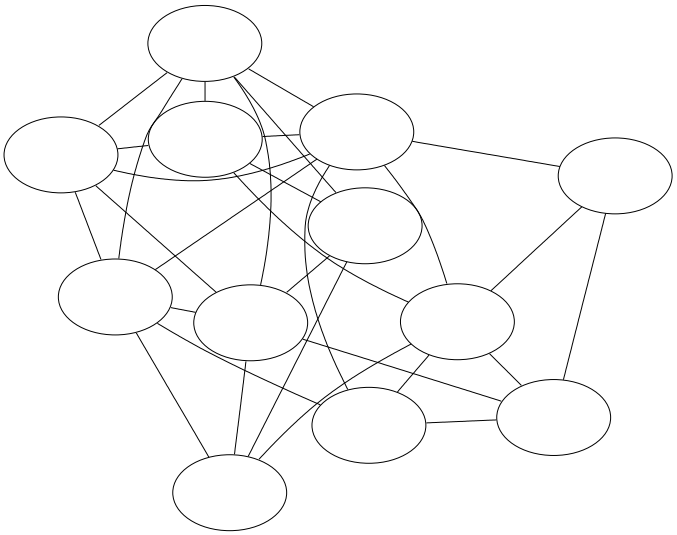
   Best-of:
   – « Vrai. Il suffit d'inclure `stdlib.h`. »

3. Qu'est-ce que Boost ?

   Correction: Boost est un ensemble de bibliothèques C++ sous une licence *open source* (Boost Software License), « soumises à un comité de relecture », qui étendent les fonctionnalités du C++. Ces bibliothèques sont le plus souvent générales (au sens de « non dédié à un domaine ») et génériques (utilisables avec tout type de données compatibles, supportant les paradigmes de programmation génériques usuels en C++, etc.). Plusieurs de ces bibliothèques ont été acceptées pour intégration dans le Technical Report 1 (TR1) du futur C++0x.
   Quelques exemples de bibliothèques Boost, utilisée dans le projet Leopard : Boost Variant, Boost Smart Pointers (`shared_ptr`), Boost Graph Library, Boost Tuple Library, et récemment Boost Lambda.

Best-of:
– « Boost est une librairie [...] »
– « Une classe. »
– « [Un] générateur de Makefile. »
– « Un outil pour la création du `configure`. »

2 Construction des Compilateurs : Allocation des Registres

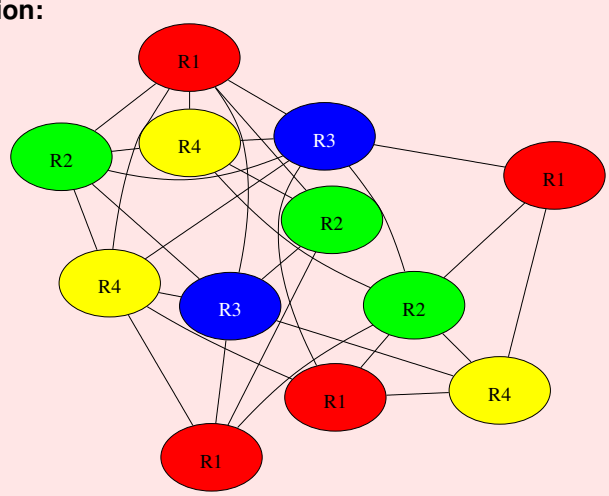


Colorer ce graphe d'exclusion mutuelle en 4 registres : R1, R2, R3, et R4.

Rendre le sujet en ayant annoté chaque nœud d'un R1, R2, R3 ou R4.

Écrire votre nom en haut.

Correction:



Merci à ceux qui ont utilisé des stylos de couleurs différentes pour remplir le graphe, cela a facilité la correction !
Attention à ne pas introduire plus de registres qu'autorisés par le sujet ! (R5, etc.)

3 Construction des Compilateurs : Support du transtypage dynamique manifeste

Le langage Leopard supporte le polymorphisme d'inclusion, et permet notamment des conversions de types *ascendantes*. C'est-à-dire, changer le type d'un objet vers l'un de ses surtypes, sans perdre d'information (son type dynamique).

On souhaite ajouter une fonctionnalité permettant de réaliser l'opération inverse, et autoriser des conversions *descendantes*. Vous connaissez déjà cette fonctionnalité en C++ via l'opérateur `dynamic_cast`. Comme on dispose déjà d'un opérateur `_cast` parmi les extensions internes du compilateur, on se propose de prendre le même mot-clef que le C++ pour implémenter notre extension, `dynamic_cast`.

Les questions sont posées dans l'ordre des stades de compilation. Certains modules tardifs nécessitent une collaboration de modules plus en amont : il est plus sain de réfléchir globalement à toutes les questions, puis de répondre dans l'ordre. Si une étape ne nécessite aucune modification, simplement le dire, et ne pas tomber verbeusement dans le piège tendu par la question.

1. Pertinence L'usage de `dynamic_cast` est déconseillé en C++ en règle générale, car il déroge à l'esprit de la programmation orientée objet. Pourquoi aurait-on envie d'encombrer notre langage avec cet opérateur décrié ?

> **Correction:** `dynamic_cast` est parfois utile localement, pour traiter des cas particuliers. Sans être une parfaite référence en la matière, le projet Leopard montre quelques usages licites de `dynamic_cast` où l'emploi d'une solution plus « propre » (par exemple, un visiteur) serait également plus fastidieuse ; par exemple, vérifier que le site de définition d'une `CallExp` est bien de type `Function`.
> Enfin, `dynamic_cast` permet l'introspection de types de données sur lesquels l'utilisateur n'a pas d'emprise – *a contrario*, un visiteur nécessite un peu d'équipement dans les classes visitées (méthodes `accept`).

> **Best-of:**
> – « Pour faire un sujet de partiel. »
> – « Parfois, le code nécessiterait bien trop de maintenance pour afin de pouvoir se passer de cette légère "hérésie". »
> – « Comme nous le disons dans l'énoncé, il peut être utile de réaliser des conversions descendantes. » (On a écrit le sujet ensemble ?)
> – « Pour se rapprocher un peu plus du C++. »
> – « Cela peut également servir à l'utilisateur naïf qui n'a pas vraiment de notion d'objet (utilisateur de type Gilbert). » (*sic*)
> – « Pour qu'un langage soit populaire, il faut pouvoir coder de manière ignoble, d'où la nécessité du `dynamic_cast`. »
> – « Parce qu'on a oublié d'implémenter la surcharge des fonctions. » (Encore une fois, attention à ne pas confondre surcharge de fonction et redéfinition de méthode, ce sont deux mécanismes qui n'ont rien à voir !)

2. Syntaxe Nous souhaitons utiliser une syntaxe similaire à celle de l'opérateur `_cast` existant pour convertir des expressions, et pouvoir écrire :

```
foo (dynamic_cast (my_var, MySubClass), 42) + 51
```

Quelle modification faut-il introduire la grammaire de notre langage pour supporter cette syntaxe ?

> **Correction:** Il suffit d'ajouter une règle de grammaire admettant une utilisation de `dynamic_cast` comme une expression :
> `<exp> ::= "dynamic_cast" "(" exp "," ty ")"`
> Attention, beaucoup de copies proposent de traiter `dynamic_cast` comme un appel de fonction (ou de primitive), mais cela n'a pas de sens : son deuxième opérande est un *type*, et non une expression.

Par ailleurs, est-ce que cette modification entraîne des difficultés, et pourquoi, le cas échéant ?

Correction: (Note préliminaire : « syntaxe » ne prend pas de « h », ni en français, ni en anglais !)
Non, cette modification n'a pas d'effet secondaire sur la grammaire. En revanche, si l'on souhaite supporter les `dynamic_cast` de l-values, il va falloir introduire une règle

```
<lvalue> ::= "dynamic_cast" "(" lvalue "," ty ")"
```

qui va entrer en conflit avec la règle précédente : il s'agit du même conflit que sur les casts non vérifiés (statiques). Ainsi, `cast (foo, string)` peut être analysé

1. comme ceci : `cast-exp` → `exp` → `lvalue (foo)`
2. ou comme ceci : `exp` → `cast-lvalue` → lvalue `(foo)`

Or c'est la seconde solution que nous voulons retenir ; pour désambiguïser, on peut utiliser les instructions `%dprec` de Bison et attribuer une priorité dynamique plus élevée à la seconde solution.

3. Syntaxe abstraite Définir la (les) classe(s) d'AST utilisée(s) pour stocker `dynamic_cast`.

Correction: Donnons les signatures, c'est bien suffisant :

```
/Ast/                 (Location loc)
  /Exp/               ()
    DyanmicCastExp    (Exp exp, Ty ty)
```

Best-of:
– « On stocke `dynamic_cast` dans un nœud `FunctionDec`. »
– « Au bindage [...] » (*sic*)

4. Liaison Est-ce que le `Binder` a un rôle à jouer ? Si oui, lequel ?

Correction: Oui : il faut que les fils des nœuds `DynamicCastExp` soient visités, notamment pour que le type destination soit lié à sa définition.

5. Typage Quelles sont les règles de typage pour `dynamic_cast` ?

Correction: Considérons l'expression **dynamic_cast** (x, Y). Notons X le type de l'expression x. Le type-checker

1. doit s'assurer que les types de X et Y sont des classes ;
2. doit vérifier le type Y est compatible avec le type X *ou vice versa* ; en effet, `dynamic_cast` peut servir à faire des conversions ascendantes ou descendantes ;
3. enfin, le nœud représentant le `dynamic_cast` est étiqueté avec le type Y.

Best-of:
– « La fonction `dynamic_cast` [...] »
– « `dynamic_cast` est typé `void*`. »
– « Une classe héritable doit devenir une classe héritante. »

6. Sémantique Wait a minute ! Une conversion dynamique peut échouer, si le type dynamique source n'est pas compatible avec le type statique cible. Considérons l'exemple suivant.

```
let
  class X {}
  class Y {} extends X {}
  var x := new X
  var y := new Y
in
  y := dynamic_cast (x, Y) /* Conversion fails! */
end
```

Dans ce cas, quel pourrait être le comportement de ce programme ?

Correction: Différentes réponses acceptables :
1. un `dynamic_cast` renvoie `nil` ;
2. une exception est levée : mais cela implique qu'il faut équiper le langage de fonctionnalités de définition et de gestion de telles exceptions, et peut-être modifier le *runtime* ;
3. le programme s'arrête (pas terrible).

Éventuellement, quel est l'impact sur les règles de typage des objets ?

Correction: La solution 1 implique un assouplissement des règles de typage des objets : `nil` doit être compatible avec tout type classe. La solution 2 nécessite le support d'un mécanisme d'exception : une telle extension (non triviale) du langage peut nécessiter l'ajout de nouveaux « types exceptions ». Enfin, la dernière solution ne requiert aucun changement vis-à-vis du typage.
Quelques erreurs notables :
– « Durant le type-checking, il faut vérifier que le type dynamique est compatible avec le type statique cible. En l'occurrence je ne vois pas où est le problème dans cet exemple... »
Le problème tient au fait que ce qui est *dynamique* se passe *après* le type-checking ! Lorsqu'un système de typage est affaibli pour autoriser un nœud à ne pas avoir un type compatible avec le type attendu – comme dans le cas présent avec l'introduction de `dynamic_cast` – le type-checker peut tout bonnement laisser passer des erreurs qui ne pourront au mieux qu'être détectées à l'exécution. Ici, tous les X peuvent ne pas être des Y, d'où erreurs éventuelles.
– Certaines copies proposent de conserver l'information de type exact à la compilation pour traiter ce problème à la compilation. Mais ce n'est pas toujours possible ! En effet, le type dynamique peut dépendre d'une action à l'exécution, et prévenir toute tentative d'analyse statique.
Dans l'exemple suivant, une telle analyse statique serait incapable de déterminer le type exact de `a`.

```
class A {}
class B {} extends A {}
class C {} extends A {}
var a : A := if getchar () = "b" then new B else new C
```

– Certaines copies proposent aussi de faire du *slicing* (« découpage » d'objet) lors de la conversion, ce qui n'est pas envisageable. D'une part on perdrait de l'information, et d'autre part ça n'a aucun sens lorsque l'on effectue une conversion *descendante*.
– D'aucuns proposent de... ne rien faire en cas d'erreur. Ce n'est pas acceptable : il nous faut un moyen de savoir si la conversion a réussi (c'est un trait de `dynamic_cast`).

Best-of:
– « Ici effectivement il y a un problème. »
– « Le programme pourrait exploser [...] »

7. Désucrage Tout comme les autres constructions orientées objet de Leopard, on souhaite désucrer `dynamic_cast` vers Tiger (le même langage, mais dépourvu d'entités OO).

On se propose de traiter un petit exemple avec... du code à trous. Considérons le code suivant :

```
let
```

```
 class Human { /* ... */}
 class Hero extends Human { /* ... */}
 var jack := new Hero
 /* Let's consider Jack Bauer as a human. */
 var agent : Human := jack
in
 /* Hey, Jack Bauer is no mere human, and he'll
    prove it by passing this conversion! */
 jack := dynamic_cast (agent, Hero)
end
```

L'exercice est de désucrer l'exemple ci-dessus, en s'aidant du code désucré (à trous) ci-après, dans un contexte simplifié :
- on fait abstraction de `Object` ;
- on ne s'intéresse pas au contenu des classe (qui est remplacé par `/* ... */`) ;
- seul le désucrage de `dynamic_cast` d'expressions nous intéresse (on ne considérera pas les `dynamic_cast` de l-values, que nous n'avons d'ailleurs pas traités dans les questions précédentes).

Ne remplir (sur votre copie) que les trous étiquetés (1) et (2) dans le code ci-dessous.

```
let
 /* Class labels. */
 var _id_Human := 1
 var _id_Hero := 2

 /* Desugared classes. They are composed of a record holding
    the actual contents of the class, and a variant able to
    store any valid concrete type for the considered (static)
    type. */
 type _contents_Human = { /* ... */}
 type Human = {
  exact_type : int,
  field_Human : _contents_Human,
  field_Hero : _contents_Hero
 }

 type _contents_Hero = { /* ... */}
 type Hero = {
  exact_type : int,
  field_Hero : _contents_Hero
 }

 /* Ctors. */
 function _new_Human () : Human =
  let
   var contents := _contents_Human { /* ... */}
  in
   Human {
    exact_type = _id_Human,
    field_Human = contents,
    field_Hero = nil
   }
  end

 function _new_Hero () : Hero =
```

```
  let
    var contents := _contents_Hero { /* ... */ }
  in
    Hero{
      exact_type = _id_Hero,
      field_Hero = contents
    }
  end

  /* Conversion routine. */
  function _cast_Hero_to_Human (source : Hero) : Human =
    Human {
      exact_type = _id_Hero,
      field_Human = nil,
      field_Hero = source.field_Hero
    }

  /* Equipment for dynamic casts. */
  /* (1) */

  var jack := _new_Hero ()
  var agent : Human := _cast_Hero_to_Human (jack)

in
  /* (2) */
end
```

Correction:

(1) Il faut introduire une routine de conversion qui vérifie que le type dynamique source est bien compatible avec le type statique cible. Ici, il n'y avait que deux classes, donc cette vérification se résumait à un choix binaire. En cas d'échec, on renvoie `nil` (si l'on décide d'adopter la solution 1 de la question 6).

```
function _dynamic_cast_Human_to_Hero (source : Human)
  : Hero =
  if source.exact_type = id_Hero then
    /* The dynamic type of SOURCE is Hero, thus the
       conversion is valid. */
    Hero {
      exact_type = _id_Hero,
      field_Hero = source.field_Hero
    }
  else
    /* The conversion failed. */
    nil
```

(2) Ici, il suffit d'appeler la routine précédente pour réaliser la conversion avant l'affectation.

```
jack := _dynamic_cast_Human_to_Hero (agent)
```

8. Langage intermédiaire Quelle modification apporter au langage intermédiaire `Tree` pour supporter `dynamic_cast` ?

Correction: On n'a besoin de rien de nouveau : le désucrage a transformé le code avec l'extension proposée vers du code sans cette extension.

9. Génération du code intermédiaire Comment modifier la génération de code intermédiaire pour supporter `dynamic_cast` ?

Correction: Pas de changement.

Best-of:
– « Avec Emacs ? »

10. Canonisation Comment modifier la traduction HIR vers LIR ?

Correction: Pas de changement.

11. Sélection des Instructions Comment modifier la traduction LIR vers assembleur MIPS ?

Correction: Pas de changement.

12. Graphe d'Interférence Comment modifier la génération des graphes de flot de contrôles, de vivacité, d'exclusion mutuelle ?

Correction: Pas de changement.

13. Allocation des Registres Comment modifier l'allocation des registres ?

Correction: Pas de changement.

Best-of:
– « Un nouveau registre permettra de faire un `dynamic_cast`. »

Vous reprendrez bien un peu de sucre ?

14. Tentative d'affectation On voudrait pouvoir disposer d'un opérateur « ?= » (*Assignment Attempt* en Eiffel), qui permet d'effectuer une opération de transtypage dynamique lors d'une affectation :

```
let
  class A {}
  class B extends A {}
  var a := new B
  var b := new B
in
  b ?= a  /* Conversion succeeds. */
end
```

Quelle est la façon la plus simple d'implémenter ça dans notre compilateur ?

Correction: Tout naturellement, on désucre vers `dynamic_cast` :

$$x \text{ ?= } y \quad \rightsquigarrow \quad x \text{ ?= } \textbf{dynamic\_cast}\ (y, X)$$

où X est le type (statique) de x, que l'on aura pris soin de conserver lors du désucrage. Pour cette raison, cette transformation ne peut se situer qu'après la vérification des types.
(Lorsque vous donnez un exemple de désucrage, merci de définir toutes les entités utilisées !)

Best-of:
– « On enlève le "?", il ne sert à rien. »

Deuxième partie

Évaluation des cours de THL, CCMP, TYLA et du Projet Leopard

Cette partie de l'examen est une enquête sous forme de QCM visant à évaluer les enseignements de THL, CCMP et TYLA, ainsi que le Projet Tiger^WLeopard. Répondez sur la feuille de QCM qui vous sera fournie lors de l'épreuve. N'y passez pas plus de dix minutes. Le temps passé à répondre à ces questions sera récompensé par trois points sur la note finale (n'oubliez pas d'inscrire votre nom sur la feuille de QCM).

Questions 1-3 (1 pt)

1. Quelle a été votre implication dans les cours (THL, CCMP, TYLA) ?
   (ne rien cocher) C'est quoi THL, CCMP, TYLA ?
   a Pas pris de notes, je découvre le livre maintenant.
   b J'ai bachoté juste avant les partiels.
   c Je vais en cours, je prends des notes.
   d Relecture régulière des notes du cours précédent.
   e J'ai approfondi le sujet par moi-même.
2. Y a-t-il de la triche dans le projet Leopard ?
   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.
3. Vous avez contribué au développement du compilateur de votre groupe :
   a presque jamais.
   b moins que les autres.
   c équitablement avec vos pairs.
   d plus que les autres.
   e pratiquement seul.

Questions 4-10 (1 pt) Leopard vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille suivante.

a Pas du tout
b Trop peu
c Correctement
d Bien
e Très bien

4. Formation au C++
5. Formation à la modélisation orientée objet et aux *design patterns*.
6. Formation à l'anglais technique.
7. Formation à la compréhension du fonctionnement des ordinateurs.

8. Formation à la compréhension du fonctionnement des langages de programmation.
9. Formation au travail collaboratif.
10. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.

Questions 11-24 (1 pt) Comment furent les tranches de `lc` (ne pas répondre à celles que vous n'avez pas faites). Répondre selon la grille suivante.

a Trop facile
b Facile
c Nickel
d Difficile
e Trop difficile

11. `LC-0`, Scanner & Parser.
12. `LC-1`, Scanner & Parser en C++, Autotools.
13. `LC-2`, Construction de l'AST.
14. `LC-3`, Liaison des noms.
15. `LC-4`, Typage.
16. `LC-5`, Traduction vers représentation intermédiaire.
17. `LC-6`, Simplification de la représentation intermédiaire.
18. `LC-7`, Sélection des instructions.
19. `LC-8`, Analyse du flot de contrôle.
20. Option `LC-E`, Calcul des échappements.
21. Option `LC-A`, Surcharge des fonctions.
22. Option `LC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
23. Option `LC-B`, Vérification dynamique des bornes de tableaux.
24. Option `LC-I`, Mise en ligne du corps des fonctions.