Votre nom :

CMP2 — Construction des Compilateurs

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

Juin 2007

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.

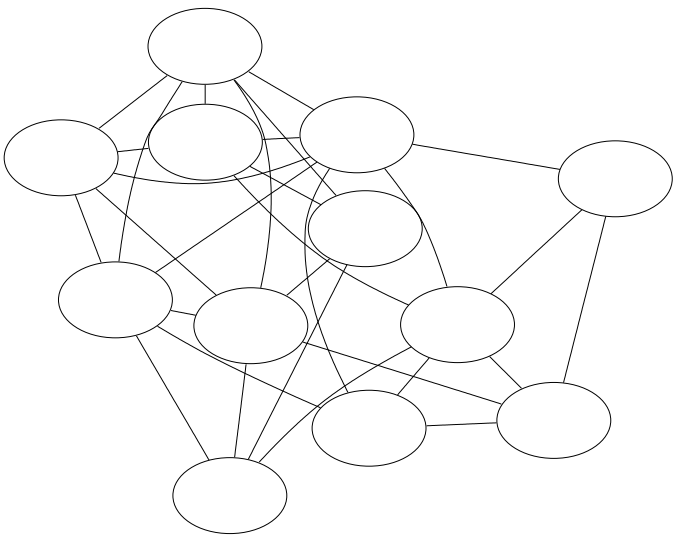
Part I
Épreuve de Construction des Compilateurs

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 ?
2. `sizeof` est une fonction de la bibliothèque standard du langage C. vrai/faux ?
3. Qu'est-ce que Boost ?

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 nœudd'un R1, R2, R3 ou R4.

Écrire votre nom en haut.

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 ?

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

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

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

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

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

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
```

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

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

10. **Canonisation** Comment modifier la traduction HIR vers LIR ?

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

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

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

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 ?

Part II
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.