Correction du Partiel de Rattrapage Compilation : Langages et Grammaires

EPITA – Promo 2006

Juillet 2004 (1h30)

Correction: Ce sujet a été écrit dans l'urgence par Clément Vasseur et Valentin David, à la demande de la direction des études, en l'absence d'Akim Demaille. Il est essentiellement composé des questions déjà posées par le passé.

Aucun document autorisé[1]

Une copie concise, bien orthographiée, dont les résultats sont clairement exposés, sera toujours mieux notée qu'une copie qui aura demandé une quelconque forme d'effort de la part du correcteur.

1 Polymorphismes

1. Qu'est-ce que le polymorphisme ? On s'attachera à donner une définition générale, et non pas liée à une forme particulière de polymorphisme, ou bien à un langage spécifique.

1. « Aucun document autorisé » signifie que ni notes de cours, livres, annales, etc. ne sont consultables pendant l'épreuve.

Correction: Plusieurs possibles :

Catégories de réponses	Récompense
La réponse se limite à la décomposisiton étymologique du mot. « polymorphisme = poly (plusieurs) + morphe (forme), qui prend plusieurs formes »	0
– Réponse très vague, difficilement compréhensible voire incomplète. – Confusion entre les mots « variable », « type », « fonction » ou « méthode », « environnement ».	1
– La réponse décrit complètement une forme de polymorphisme. – La réponse décrit brièvement plusieurs formes de polymorphisme. – La réponse formule le concept de polymorphisme en programmation. « le principe du polymorphisme est de choisir la méthode appropriée selon le type de la variable. »	2
– La réponse formule de manière claire et concise le concept de polymorphisme. – La réponse cite et décrit correctement plusieurs formes de polymorphisme. « C'est la possibilité d'avoir plusieurs entités (fonctions, classes) portant le même nom mais ayant des attributs différents, sans entraîner aucune ambiguité dans le comprotement du programme. »	3

2. Soient `a` et `b` deux variables Tiger. Donner une expression Tiger dont le code effectif (c'est-à-dire celui qui sera exécuté) dépend du type de `a` et `b`.

Correction: `a < b` donnera lieu à du code très différent selon que `a` et `b` sont des entiers, ou des chaînes de caractères (où il faut implémenter `strcmp`).
C'est la surcharge, "overloading".

Catégories de réponses	Récompense
La réponse n'évoque pas les opérateurs de comparaison sur les chaînes et les entiers.	0
La réponse évoque vaguement le cas.	1
La réponse semble décrire le cas, mais son expression ou l'explication qui suit est douteuse.	2
La réponse montre le cas mais n'explique rien. « `a < b` »	3
La réponse décrit la surcharge des opérateurs de comparaisons entre les chaînes et les entiers. « L'expression `a < b` donnera lieu à la génération d'un code différent selon que a et b sont des chaînes de caractères (appel à `strcmp`) ou que a et b sont des entiers (comparaison simple) »	4

3. Comment le compilateur détermine-t-il la version du code à générer ?

Correction: Le contrôle de type doit avoir été effectué, de façon précisément à déterminer le type de `a` et `b` dans l'exemple précédent. En se basant sur ce typage, le générateur de code produira celui de $<_{\texttt{int}}$ ou celui de $<_{\texttt{string}}$.

Catégories de réponses	Récompense
La réponse n'évoque pas le typage.	0
La réponse révèle une confusion entre vérification du type et sélection sur le type ; le point attribué correspond au bénéfice du doute. « lors du type-checking, le compilateur vérifie la correspondance des types et détermine si le code peut être généré correctement. »	1
La réponse est correcte mais trop floue, confuse ou incomplète. « Grâce au typage des variables. »	2 à 3
La réponse parle bien du typage et du choix de la fonction à appeler.	4

Best-of:

« la version du code à générer est déterminée par les extensions de fichiers. »

« version est donnée dans le `configure.ac` »

« en fonction du nombre de fois que l'on a compilé et changé la grammaire. »

« le compilateur détermine la version avec `bison -dv`. »

« le compilateur détermine la version du code à générer à partir d'une variable du fichier `configure.ac` qui contient le numéro de version, le nom et l'email des auteurs. »

« le compilateur détermine la version du code à générer, soit par les dates de modification de fichier (bof... mais ca marche), sinon en analysant le code objet (me semble-t-il...) »

« A la compilation, le code est inliné les définitions sont remplacées par le code. »

« A partir des bibliothèques »

« Le compilateur se base sur l'architecture et le type de l'OS du PC pour déterminer la version du code à générer »

4. Dans l'expression C `sin (51)`, quel autre mécanisme de polymorphisme intervient ?

Correction: Certains types sont automatiquement promus, comme par exemple ici, l'entier `51` sera automatiquement converti en un `double`.

Catégories de réponses	Récompense
– La réponse est incongrue. – La réponse parle de ce que le C n'a pas, comme la surcharge en entrée.	0
La réponse est confuse. « conversion de type == overloading »	1 à 2
La réponse se limite à : « Le cast. »	3
– La réponse parle de promotion (très bien !) – La réponse parle de transtypage : l'`int` permet de créer une 'instance' de `float`. – La réponse parle de coercion : l'`int` est considéré comme type builtin, et se déduit directement en `float` (il n'y a pas à proprement parler de création d'une "instance" `float`).	4

Best-of:

« en C `sin` prend en argument un réel. Mais un réel dans notre cas ici peut être exprimé en degrés ou en radians »

« Dans cette expression, on voit ici une fonction qui prend comme argument un entier et qui retourne un réel. Ceci est une autre forme de polymorphisme »

« Dans l'expression `sin(51)`, on parle de typage statique. »

« Polymorphisme de type de sortie pour la fonction `sin` »

« Polymorphisme paramétrique »

« Le polymorphisme d'inclusion géré au moment de l'exécution »

5. Voyez-vous en Tiger une valeur qui puisse avoir plusieurs types ?

Correction: il fallait bien sûr expliquer le cas de `nil`.

Catégories de réponses	Récompense
La réponse est incongrue. « n'importe quelle variable peut avoir plusieurs types » « oui, sinon le type-checking ne sert à rien »	0
– La réponse évoque vaguement `nil`. « un record peut être de type [son type] et `nil` » – La réponse est intelligente : « alias sur type »	2 à 3
« nil peut prendre plusieurs types » « nil est compatible avec tous les types cartésiens. »	4

6. Les langages orientés objets introduisent une forme de polymorphisme. Comment l'appelleriez-vous ?

Correction: Polymorphisme d'héritage, ou d'inclusion, voire `virtual`.

Catégories de réponses	Récompense
– La réponse est incongrue. « type void en C++ » – La réponse oublie que Java est orienté objet, et que ML ne l'est pas. « polymorphisme paramétrique » – La réponse oublie le reste du sujet. « surcharge »	0
La réponse est confuse mais l'idée est là. « opération polymorphe sur une hiérarchie de classes »	1
La réponse évoque le polymorphisme de classe, d'héritage ou d'inclusion.	4

7. À quel mot-clé est-il associé en C++ ?

Correction: `virtual`.

Catégories de réponses	Récompense
La réponse est incongrue	0
– La réponse uniquement donne `virtual` uniquement pour C++ – ou pour simula	2
– La réponse donne `virtual` pour C++ – et simula.	4

Best-of:

« generic, template », « namespace », « overriding », « en C et en Simula on utilise le cast », « NULL », « : », « : : »

« en C++, le mot clé 'able' nous permet de définir par exemple qu'un moustique est 'mouche-able' »

8. Sur quelle forme de polymorphisme s'appuie largement STL ?

Correction: Polymorphisme paramétrique, lié à la généricité. En C++, c'est lié au mot-clé `template`.

Catégories de réponses	Récompense
La réponse est incongrue. « le dynamic_cast », « polymorphisme de classe », « coercion », « la STL s'appuie sur les conteneurs »	0
La réponse est confuse. « template c'est à dire polymorphisme de classe » « polymorphisme de substitution c'est à dire template »	1
La réponse oublie que `template` est un mot-clé du C++. « les templates. »	2
La réponse parle de « polymorphisme de type », « polymorphisme paramétrique » ou de « généricité »	4

2 Techniques C++ : "Traits"

1. À quoi sert le mot-clé `typename` ?

   Correction: À aider le compilateur à comprendre que ce que l'on extrait d'un patron (pas instantié) est un type.
   Par ailleurs, mais ce n'est pas ce qui m'intéressait ici, il sert aussi pour « typer » les paramètres en spécifiant qu'on attend un type (et d'ailleurs pas nécessairement un nom de type : on peut passer `std::list <int>` par exemple).

2. Que sont les "traits" ?

   Correction: Des fonctions dont les arguments sont des types. Elles sont donc évaluées à la compilation.

3. À quoi peuvent servir les "traits" ?

   Correction: À apprendre des choses sur un type :
   – pour les types numériques, les caractéristiques (le max, min etc.).
   – s'il est const
   – sa version const
   – sa version pas const
   – s'il est pointeur
   – ...

4. Écrire un "traits" qui pour tout type "retourne" son type souche non pointeur, i.e.

   `int` → `int` `int*` → `int` `int****` → `int`

Correction:

```
template <typename T>
struct unpointer
{
  typedef T type;
};

template <typename T>
struct unpointer <T *>
{
  typedef typename unpointer <T>::type type;
};

// ============================================== This part was not asked for.

#include <iostream>
#include <typeinfo>

int
main ()
{
#define TEST(In, Out)                                              \
  {                                                                \
    std::cout << typeid (In).name () << ":\t"                      \
              << typeid (unpointer<In>::type).name() << " vs. "    \
              << typeid (Out).name () << " = "                     \
              << (typeid (unpointer<In>::type) == typeid (Out))    \
              << std::endl;                                        \
  }

  TEST (int,  int);
  TEST (int*, int);
  TEST (int****, int);
  TEST (int*, float);
}
```

5. Qu'est-ce qu'un objet fonction ?

   Correction: Un objet qui dispose d'une méthode `operator()`, i.e., qui a une méthode dont l'invocation ressemble à l'appel d'une fonction.

3 Compilation : Boucles Expressions

On étudie la possibilité de considérer les boucles comme des expressions retournant une valeure, et non plus simplement des instructions. On prendra Tiger comme exemple concret de langage souche.

Correction: Il semblerait que la première question n'ait même pas été la question 0, mais tout simplement comprendre ce paragraphe. Par « expression » j'entendais ce que j'ai toujours dit : « qui a une valeur », précisément par opposition à « instruction ». Le fait que dans notre implémentation de tc des choses telles que les boucles sont des `Exp` est sans rapport : ce n'est qu'un détail interne lié au fait qu'on ait choisi de laisser le contrôle de type faire la différence plutôt que le parseur. Mais fondamentalement le langage Tiger a bien des instructions (comme `break`) et des expressions (comme `51`).
Ce problème consiste donc à réfléchir à la possibilité de créer des boucles qui retournent une valeur.

1. Quelles sont les trois constructions dans l'AST qui sont **en rapport** avec les boucles ?

   Correction: On a `WhileExp`, `ForExp`, et surtout `BreakExp`.

2. Dans quel cas est-il manifestement difficile pour une boucle non interrompue de retourner une valeur ?

   Correction: Si on ne rentre jamais dedans.

3. Proposer une extension de syntaxe des boucles qui permettrait de spécifier le comportement à tenir dans ces cas-là. On suggère fortement de s'inspirer d'une autre partie du langage qui pose un problème similaire. En particulier, éviter l'introduction de nouveau mot clé.

   Correction: On peut adjoindre un `else`, comme on le fait pour les « `if-then` à l'intérieur desquels on ne rentre pas ».

4. Votre extension est-elle susceptible de provoquer de nouvelles ambiguïtés dans la grammaire ? Si oui, définir la bonne lecture d'une telle phrase ambiguë.

   Correction: Bien sûr ! Pour exactement les mêmes raisons que le `else` de if. Et même on peut mélanger le « danging-else » d'une boucle avec celui d'un if :
   `if e then for i := 0 to 51 do i else -1`
   À qui appartient le `else` ?
   Bien sûr on l'attachera au plus proche, qu'il s'agisse d'une boucle ou d'un `if`.

5. Quelle autre fonctionnalité citée en question 1 compromet le concept de "valeur" d'une boucle ?

   Correction: `break`.

6. Proposer, si nécessaire, une modification de la syntaxe Tiger de cette fonctionnalité pour supporter les boucles expressions.

   Correction: Il nous faut pouvoir "retourner" une valeur : `break` peut avoir un argument.

7. Proposer des règles de typage pour toutes ces constructions.

   Correction: Comme pour le `if` : les deux « branches » de la boucle (i.e., son corps et son `else`) doivent avoir des types compatibles (i.e., égaux, ou l'un est un enregistrement et l'autre est `nil`), et ceci constitue le type de la boucle. De plus, les `break` doivent avoir un argument compatible avec ce type.

8. Proposer une implémentation très économe (mais naïve) du support des boucles expressions dans un compilateur comme `tc`. En quoi est-elle naïve ?

Correction: On peut facilement dé-sucrer de telles boucles :

```
while <cond>
  <body>
else
  <default>
donne
```

```
let
  var res := <default>
in
  while <cond>
    res := <body>;
  res
end
```

en remplaçant les `break val` par `(res := val; break)`. On procède de façon similaire pour les boucles `for`. On retombe bien alors sur du Tiger traditionnel, et l'on peut même reposer sur le contrôle de type de celui-ci pour contrôler notre extension. Les messages d'erreur seront cependant peu judicieux.
Il s'agit donc d'introduire un désucrage juste après le parsing (voire pendant si vous êtes ambitieux).

Cela dit, c'est trop naïf, et c'était bien le sens de la question préliminaire : en présence d'effets de bord, ça n'est pas correct. La fonction de la question préliminaire était précisément de souligner ce danger.
Par exemple l'application de cette transformation à l'exemple suivant est condamnée à l'erreur.

```
let
  var cond := 1
in
  while cond
    cond := 0
  else
    (print ("Hello, World!\backslash{}n"); 51)
end
```

Il faut plutôt aller vers quelque chose comme :

```
let
  var init := 0
  var res  := ???
in
  while <cond>
    (init := 1;
     res  := <body>);
  if not init then
    res := <default>;
  res
end
```

Mais on butte contre le fait que ce n'est pas du langage Tiger valide : la variable `res` n'est pas correctement déclarée. On ne pourra donc pas se permettre cette transformation sans aide de la part du contrôle de type. Bien sûr l'approche la plus complète serait de préserver cette structure jusqu'à T5, puis le traduire à ce moment-là.

Annexe : Grammaire Tiger

We use Extended BNF, with '[' and ']' for zero or once, '(' and ')' for grouping, and '{' and '}' for any number of repetition including zero.

```
program ::= exp

exp ::=
  # Literals.
    'nil'
  | integer
  | string

  # Array and record creations.
  | type-id '[' exp ']' 'of' exp
  | type-id '{'[ id '=' exp { ',' id '=' exp } ] '}'

  # Variables, field, elements of an array.
  | lvalue

  # Function call.
  | id '(' [ exp { ',' exp }] ')'

  # Operations
  | '-' exp
  | exp op exp
  | '(' exps ')'

  # Assignment
  | lvalue ':=' exp

  # Control structures
  | 'if' exp 'then' exp ['else' exp]
  | 'while' exp 'do' exp
  | 'for' id ':=' exp 'to' exp 'do' exp
  | 'break'
  | 'let' decs 'in' exps 'end'

lvalue ::= id
  | lvalue '.' id
  | lvalue '[' exp ']'
exps ::= [ exp { ';' exp } ]
decs ::= { dec }
dec ::=
  # Type declaration
   'type' id '=' ty
  # Variable declaration
  | 'var' id [ ':' type-id ] ':=' exp
  # Function declaration
  | 'function' id '(' tyfields ')' [ ':' type-id ] '=' exp

# Types
ty ::= type-id
   | '{' tyfields  '}'
   | 'array' 'of' type-id
tyfields ::= [ id ':' type-id { ',' id ':' type-id } ]
type-id ::= id

op ::= '+' | '-' | '*' | '/' | '=' | '<>' | '>' | '<' | '>=' | '<=' | '&' | '|'
```