Correction du Compilation, Langages de Programmation

EPITA – Promo 2006 – **Tous documents autorisés** *

Avril 2004 (1h30)

Le sujet a été écrit 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.

Les résultats ne doivent pas être balancés comme « évidents », sous peine de disqualification pour tentative de bluffage. Néanmoins, une argumentation informelle mais convaincante, sera souvent suffisante.

1 Typologie des Langages de Programmation

1. Apparier chaque auteur avec son langage :

   a. Alan Kay
   b. Andrew Appel
   c. Bjarne Stroustrup
   d. Denis Ritchie
   e. John Backus
   f. Kristen Nygaard
   g. Niklaus Wirth
   h. Ole-Johan Dahl

   1. Ada 83
   2. C
   3. C++
   4. Pascal
   5. Simula
   6. Smalltalk
   7. Tiger

   Correction: Je dois avouer avoir fait une erreur en simplifiant cette question qui à l'origine portait sur Ada et Lisp. Malheureusement, j'ai retiré FORTRAN au lieu de retirer Ada 83.

Who	What
Alan Kay	Smalltalk
Andrew Appel	Tiger
Bjarne Stroustrup	C++
Denis Ritchie	C
Jean Ichbiah	Ada 83
John Backus	FORTRAN, FP
John McCarthy	Lisp
Kristen Nygaard	Simula, Beta
Niklaus Wirth	Algol-W, Euler, Pascal, Modula-2, Oberon
Ole-Johan Dahl	Simula, Beta

2. Que sont les multiméthodes ?

*"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.

Correction: Sélection à l'exécution de la bonne fonction selon le type dynamique de n'importe quel argument, et non pas seulement l'argument 0 (`this`). C'est un support à l'exécution de ce que la surcharge permet déjà de faire à la compilation.

3. Citer quelques cas où les multiméthodes offrent une facilité d'implémentation.

   Correction: Un premier gain, simple, c'est de ne plus être obligé « d'être dans la classe » pour pouvoir avoir la sélection dynamique. Citons comme exemple la classique méthode `print` qui devient une fonction extérieure. En fait, les Visiteurs n'ont pas besoin des méthodes `accept` pour fonctionner. Les Visiteurs simulent les multiméthodes.
   Par ailleurs, on a souvent besoin de sélection selon deux arguments : par exemple l'arithmétique entre `Numbers`, où `Number` est une classe virtuelle, ou bien encore, exemple célèbre, la recherche de l'intersection entre deux `Shapes`, où `Shape` est là encore une classe abstraite.

4. Citer un langage supportant les multiméthodes.

   Correction: CLOS est celui dont j'ai le plus parlé, mais il y a aussi Perl 6 qui aurait une implémentation.

5. En C++, à quelle technique fait-on appel pour simuler les multiméthodes ?

   Correction: Les VISITORS.

6. En Eiffel, lorsqu'une méthode est redéfinie, qu'advient-il de ses préconditions ?

   a. elles ne sont pas modifiables
   b. elles peuvent être affaiblies
   c. elles peuvent être renforcées
   d. elles sont librement modifiables

   Correction: Pas besoin de connaître Eiffel pour répondre ! Bien sûr, 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.

7. En Eiffel, lorsqu'une méthode est redéfinie, qu'advient-il de ses postconditions ?

   a. elles ne sont pas modifiables
   b. elles peuvent être affaiblies
   c. elles peuvent être renforcées
   d. elles sont librement modifiables

   Correction: Pour les mêmes raisons, les postconditions peuvent être plus strictes.

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 Souvenirs, souvenirs... Analyse Syntaxique

Écrire un fichier `bison/yacc` *complet* qui analyse une liste de zéro ou plusieurs entiers (token `NUM` de type `int`) séparés par des virgules, et à la fin la stocke dans la variable globale `the_list` de type liste d'entiers.

Ce fichier `parselist.yy` doit pouvoir être utilisable *directement* ; prendre garde :
– aux `#includes` nécessaires
– déclarer les variables et fonctions (`yylex`) externes dont on dépend
– définir `void yyerror (const char *)`
– définir les types de valeur dont on aura besoin (`%union`)
– déclarer les types des tokens (`%token`) et des non terminaux (`%type`) nécessaires
– faire une grammaire qui compile sans conflits

– faire une grammaire efficace
– expliquer pourquoi votre réponse est “efficace”.

Ni l’analyseur lexical, ni `main` ne sont demandés.

Correction: Voici le parseur.

```
%{          // parselist.yy -*- C++ -*-
# include <list>

  typedef std::list <int> ints_type;
  extern ints_type *the_list;
  int yylex ();

  void
  yyerror (const char *msg)
  {
    std::cerr << msg << std::endl;
  }
%}

%union
{
  int ival;
  ints_type *ints;
}

%token <ival> NUM
%type  <ints> list list.1

%%
list:
  /* empty */    { the_list = new ints_type; }
| list.1         { the_list = $1; }
;

list.1:
  NUM           { $$ = new ints_type; $$->push_back ($1); }
| list ',' NUM  { $$ = $1; $$->push_back ($3); } //$
%%

//================================================ This part was not asked for.
#include <iostream>
#include <iterator>
#include <algorithm>

ints_type *the_list;

int
main ()
{
  yyparse ();
  std::copy (the_list->begin (), the_list->end (),
             std::ostream_iterator <int> (std::cout, "\n"));
  return 0;
}
```

Et pour information, voici le scanner.

```
%{          // scanlist.ll -*- C++ -*-
//================================================ Provided, but not asked for.
# include <list>
  typedef std::list <int> ints_type;
# include "parselist.hh"
%}
%%
[0-9]+      yylval.ival = atoi (yytext); return NUM;
,           return *yytext;
[ \t\n]+    /* skip */;
%%
```

4 Compilation : Boucles Expressions

On étudie la possibilité de considérer les boucles comme des expressions, 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.

0. Préliminaire : Les `SeqExp` peuvent poser des problèmes de compilation, ce pourquoi on cherchera à les éliminer. Expliquer cependant pourquoi la démarche naïve consistant à remplacer `v := (s; e)` par `s; v:= e` n'est pas toujours correcte (`v` désigne une variable, `s` une instruction, et `e` une expression).

   Correction: J'ai voulu aller trop vite en rajoutant au dernier instant cette question sensée vous aider, et du coup... j'ai donné un cadre de pensée où il n'y a pas de différence. Il aurait mieux valu que je propose de réfléchir à `e1 + (s; e2)` vs. `(s; e1 + e2)` où l'on voit bien que si `s` influence la valeur de `e1`, alors les deux ne sont pas équivalents. Par exemple `a + (a := a + 1; 1)`.
   Si au lieu de « `v` désigne une variable » j'avais écrit « `v` désigne une l-value », alors on pouvait suggérer `a[i] := (i := i + 1; 42)` qui est différent de `(i := i + 1; a[i] := 42)`.

1. Quelles sont les 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.