Correction du Partiel CMP1-TYLA

EPITA – Promo 2009 – **Tout document autorisé**

Avril 2007 (1h30)

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

1 Incontournables

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

1. Quelle est le type (de Chomsky) du langage engendré par $S \to aX \quad S \to b \quad X \to Sc$

   Correction: Ce langage est $a^n bc^n$ bien connu pour être hors-contexte, et non rationnel.

2. Combien existe-t-il de mots de m lettres prises dans un alphabet de taille n ?

   Correction: n^m, on a m fois le choix parmi n caractères.

   Best-of: Il y a eu beaucoup de réponses inventives. Parmi les résultats, on pouvait lire :

   - n
   - n^2
   - $n - m + 1$
   - $n!$
   - $m!$
   - $(n - m)!$
   - $m \times n$
   - $n \times (m - n)$
   - $\frac{m}{n}$
   - m^n
   - $m^n + 1$
   - m^{n-1}
   - C_n^m,
   - $C_m^n = \frac{n!}{m!(n-m)!}$,
   - $\begin{cases} C_n^m \times m! & \text{si} \quad m < n \\ m! & \text{sinon} \end{cases}$
   - $\begin{cases} 0 & \text{si} \quad m > n \\ A_n^m & \text{si} \quad m < n \end{cases}$
   - m parmi n
   - un nombre fini
   - une infinité

3. La récurrence à droite est-elle possible en LALR(1) ?

   Correction: Oui, bien entendu ! On préconise la récurrence à gauche pour soulager la pile des parsers LALR(1), mais la récurrence à droite est tout à fait valide.

   Best-of:
   - Oui, elle est même conseillée (*sic*). On pourrait avoir des conflits shift/reduce ou reduce/reduce et faire exploser la pile de tokens.

4. Qu'est-ce que Yacc ?

   Correction: Un générateur de parsers.

   Best-of:
   - Yacc : Yet Another C Compiler (*sic*).
   - Un programme qui à partir de règle(s) engendre un lexer.
   - Yet Another Compiler. C'est un scanner qui en général est associé avec le parser nommé Bison.

2 Typologie des Langages

1. Parmi les polymorphismes ci-dessous, lesquels sont disponibles en C ISO 1999 ?
   - polymorphisme paramétrique
   - polymorphisme de surcharge (également appelé polymorphisme *ad hoc*)
   - polymorphisme de coercition (*coercion* en anglais)
   - polymorphisme d'inclusion
2. Même question, mais en C++ ISO 2003.

Correction: Rappelons brièvement les définitions des différents polymorphismes évoqués dans l'énoncé. Un polymorphisme est un trait d'un langage de programmation permettant à une entité du dit langage d'être appliquée ou vue de plusieurs façons. Notamment, un code tirant parti d'un polymorphisme donné peut avoir la propriété d'être utilisable avec plusieurs types de données, ou fournir plusieurs types de données. Les différents types de polymorphismes permettent souvent la généricité (statique ou dynamique), l'abstraction, un meilleur découplage interface/implémentation, une extensibilité du code (plus ou moins grande), etc.

- Le *polymorphisme de coercition* est la capacité d'un type à se faire passer pour un autre : il s'agit d'une conversion de type implicite (une conversion de type explicite est appelée « *cast* » en anglais). C'est la seule réelle forme de polymorphisme en C. La coercition n'agit donc que sur les types, et est réalisée statiquement.
- Le *polymorphisme ad hoc* ou *surcharge* est la possibilité de donner le même nom à des routines différentes, qui se distinguent alors par leur signature. Il s'agit d'un mécanisme statique : le compilateur effectue la sélection (*dispatch*) de la bonne fonction au site d'appel, et effectue la liaison à la compilation. En C++, cette sélection s'appuie sur le nom de la fonction, le nombre d'arguments, leurs types et s'il s'agit d'une méthode, la constance éventuelle de l'objet-cible. Il n'y a pas de polymorphisme de surcharge général en C ; il existe cependant certains cas particuliers (comme les fonctions trigonométriques en C99) assimilables à de la surcharge. Lorsqu'ils étaient mentionnés (pour le C), la réponse a été considérée juste ; dans tous les autres cas, elle a été considérée fausse.
- Le *polymorphisme paramétrique* permet l'écriture d'entités polymorphes vis-à-vis d'un paramètre *statique*. Il est souvent utilisé comme support du paradigme de *généricité statique*. En C++, il concerne tant les fonctions et méthodes que les objets, car ce langage permet d'écrire des routines et des types paramétrés via le mot-clef `template`. Et il s'agit d'un mécanisme complètement statique : les routines et types sont instanciés[a] à la compilation, le plus souvent automatiquement (à partir du site d'utilisation), mais il est également possible de le faire explicitement.
  Là encore, le C ne dispose pas polymorphisme paramétrique, même si le préprocesseur permet d'en obtenir une forme faible. Les réponses qui y faisait explicitement mention pour justifier la présence de polymorphisme paramétrique en C ont été comptées justes.
- Enfin, le *polymorphisme d'inclusion* est celui induit par la relation « EST UN » entre objets (généralisation), et la faculté de redéfinir des méthodes polymorphes (*virtuelles* en C++), dont la sélection est faite à l'exécution (liaison retardée, *dynamic dispatch*). Il concerne tant les types que les méthodes, car il permet à l'exécution d'utiliser différents types (concrets) là où un type (abstrait) est attendu, et autorise une méthode à avoir un comportement différent selon le type dynamique de sa cible.
  Il n'y a pas de classe ni d'objet en C, donc pas de polymorphisme d'inclusion.

a. Hé oui, il s'agit du même verbe que pour la création d'objets ; attention à ne pas confondre leurs usages !

3. Classer ces quatre types de polymorphismes évoqués dans les deux questions précédentes dans les cases du tableau ci-dessous (à recopier sur votre copie).

	résolution statique	résolution dynamique
concerne les types/classes		
concerne les fonctions/méthodes		

Correction: Le tableau s'écrit naturellement d'après les explications de la réponse précédente.

	résolution statique	résolution dynamique
concerne les types/classes	$\mathcal{P}$, $\mathcal{C}$	$\mathcal{I}$
concerne les fonctions/méthodes	$\mathcal{P}$, $\mathcal{S}$	$\mathcal{I}$

avec

$\mathcal{C}$: polymorphisme de coercition — $\mathcal{I}$: polymorphisme d'inclusion

$\mathcal{S}$: polymorphisme de surcharge — $\mathcal{P}$: polymorphisme paramétrique

Attention à ne pas confondre surcharge et polymorphisme d'inclusion : beaucoup emploient le terme « méthode surchargée » pour dire « méthode redéfinie ».

4. On rappelle que la *mise en ligne* du corps d'une fonction (*inlining*) est une optimisation consistant à remplacer un appel de fonction ou de méthode par le corps de celle-ci au site d'appel (en remplaçant les arguments formels par les arguments effectifs).

   En C++, il existe plusieurs situations dans lesquelles une fonction (ou méthode) ne peut être mise en ligne ; citez-en deux. (On ne s'attache pas ici à la qualité de l'optimisation, donc une réponse telle que « le compilateur refuse l'inlining car la fonction à mettre en ligne est trop grosse » n'est pas pertinente).

   Correction: Au choix :
   - la définition de la fonction n'est pas visible depuis le site d'appel (c'est-à-dire, dans la même unité de compilation) ;
   - la fonction candidate à la mise en ligne est récursive (directement ou indirectement) : la mise en ligne ne peut avoir lieu, car il s'agirait d'un processus sans fin. (Bien entendu, on pourrait proposer de limiter la profondeur d'inlining à un certain nombre d'appels récursifs, mais ça commence à devenir trop subtil pour une question qui se voulait initialement simple !)
   - on ne peut mettre en ligne une méthode polymorphe (c'est-à-dire une fonction membre virtuelle en C++), car sa liaison est effectuée à l'exécution, alors que la mise en ligne est une opération effectuée à la compilation ;
   - un peu pour les mêmes raisons, on ne peut mettre en ligne une fonction appelée via un pointeur sur fonction ;
   - enfin, il se trouve que certains compilateurs refusent de mettre en ligne une fonction dès lors qu'elle accepte un nombre variable d'arguments (opérateur « `...` » du C++). Sans être impossible, ce choix d'implémentation est compréhensible car le passage et la récupération d'arguments variables s'effectuent via une manipulation du cadre de pile, dont la traduction lors de la mise en ligne pourrait être non triviale.

   Parmi les réponses données, on trouve souvent l'impossibilité de mettre en ligne une fonction paramétrée, alors qu'il s'agit le plus souvent d'un cas favorable, étant donné qu'un cas usuel d'utilisation est de mettre à disposition la définition des templates au(x) site(s) d'utilisation(s).

Best-of:
- Les fonctions dont les paramètres changent à l'exécution.
- Les fonctions public (*sic*).
- La fonction a des effets de bord.
- Multiples évaluations.
- Modification de priorité.
- On utilise `g++` avec `-fno-inline`.
- Les constructeurs et destructeurs ne peuvent pas être inlinés.
- Les opérateurs ne sont pas inlinables.
- Le corps de la fonction utilise trop de registres processeur.
- La fonction est dans un `namespace`.
- On ne peut mettre une fonction en ligne si on utilise les templates ou que l'on est pas connecté à Internet.
- Peut-être les fonctions / méthodes `const`.
- On ne peut faire l'inlining d'une fonction ou d'une méthode privée, ni d'une fonction qui retourne des paramètres.
- Lorsque les arguments ne sont pas d'un type de base du langage [...].
- On ne peut pas faire [d'inlining] dans le cas de fonctions qui appellent des fonctions statiques.
- Lorsque la fonction contient des macros.

5. *(Culture générale)* Quel informaticien célèbre est décédé le 17 mars 2007 ? Citer une de ses réalisations. (Bien entendu, vous n'avez pas le droit au coup de fil à un ami pour cette question.)

Correction: Il n'était nul besoin d'avoir regardé la télévision pour connaître la réponse à la question ; lire `epita.cours.compile` suffisait ! Il s'agissait de John Backus (1924-2007). On lui doit les langages FORTRAN et FP ; et la notation BNF (*Backus-Naur Form)*, sensiblement améliorée par Peter Naur.

Best-of:

- Douglas Gregor[a]
- l'Abbé Pierre
- le créateur de Perl
- Richard Stallman, Emacs
- l'inventeur du C++
- Faure, l'EBNF[b]
- Le père de Smalltalk, Alan Kay
- Moore
- Bill Gates
- Saint Patrick
- David Keyboard, à qui on doit l'invention du clavier QWERTY
- Le créateur d'Appel (*sic*)
- Andrew Appel, pour Tiger
- Richard M. Stallman ; réalisation : le C
- Obi-Wan Kenobi
- Gary Coleman, pour son ouvrage *Different Strokes to Make a (*sic*) Software*
- Bertrand Meyer (pour Eiffel)
- Nom inconnu. Il a inventé le disque dur.
- Babel
- (Le (dernier (programmeur (Lisp))))
- l'informaticien s'appelait . . . il a réalisé NetBSD
- Donald Knuth, qui a réalisé LaTeX(*sic*)
- Antoinne Zürchmayer[c]
- le « papa » de Pascal
- Cela fait un bout de temps que je n'ai pas revu sigour_b
- Ritchie, papa du C
- Forty Two (*sic*) rendu célèbre par Douglas Adams dans *Le guide du voyageur [inter]galactique*
- Pas Bill Gates, car ce n'est pas un informaticien . . .
- John Fortran, le créateur entre autres du FORTRAN
- J'hésitais entre Claude François et John Backus [. . .]
- Djisktra (*sic*)
- « Zero Cool » qui allume les fenêtres en façade d'immeuble à la fin de *Hackers*[d].
- Barry White
- Il me semble que c'est l'inventeur du KOBOL (*sic*).
- L'inventeur du langage Ada.
- K. N. King, paix à son âme[e].
- Larry Wall qui a inventé le Perl
- Alan Turing
- Pour tenter ma chance [. . .] Bjarne Stroustrup, il a créé le langage C++
- Au pif, Niklaus Wirth
- Le pape Jean-Paul II

a. Un développeur de Boost et de ConceptGCC, participant au *C++ Standards Committee*.
b. Je pense que l'étudiant voulait citer *Naur*.
c. 0 hit dans Google.
d. Voir http://www.imdb.com/title/tt0113243/.
e. Voir http://knking.com/ et http://www2.gsu.edu/~matknk/.

3 Construction des Compilateurs

3.1 Mise en bouche

1. Qu'est-ce que le sucre syntaxique ?

Correction: Il s'agit de constructions syntaxiques supplémentaires proposées par un langage n'apportant rien de plus au niveau de sa syntaxe abstraite, mais permettant des variantes d'écriture. Le sucre syntaxique apporte souvent simplicité, lisibilité, raccourcis d'écriture, confort, etc. Il peut permettre l'expression de nouveaux concepts (la surcharge de fonctions peut être considérée comme telle, par exemple).

Best-of:
- Par exemple, `while` est du sucre syntaxique.
- Exemple : le `if` est un sucre syntaxique.
- C'est du sucre à base de pétrole qui ne fait pas grossir.
- Le sucre syntaxique est une manière de rendre un langage beaucoup plus buvable [. . .].

2. Qu'est-ce que le désucrage ?

Correction: Le désucrage est l'opération qui consiste à remplacer les éléments de sucre syntaxique par des constructions de syntaxe de base (« non sucrées »).

3.2 Cassons du sucre sur le dos du félin

Dans les questions suivantes, on se propose de traiter quelques-uns des aspects du désucrage des constructions ayant trait à l'objet dans le langage Leopard. On rappelle que le traitement des dites structures objets dans `lc` passe par une transformation (désucrage) du langage autorisant ces éléments, à un sous-ensemble de Leopard n'en disposant pas, qu'on appellera *Tiger*.

1. Citer les 5 phases d'une partie frontale d'un compilateur comme `lc`.

   Correction: (i) scanner, (ii) parser et construction de l'AST, (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

   Best-of:
   – « scannage »
   – Lexing, parsing, construction de l'AST, binding, segfaulting.
   – placement des cadres

2. Où situer la phase de désucrage de « Leopard avec objets » vers « Leopard sans objets » (i.e. Tiger) ? Justifiez votre réponse.

   Correction: Avant toute chose, il est bon de préciser que le désucrage évoqué traduisait le programme vers un langage dépourvu de constructions objet. À ce titre, la transformation
   `class` c `[extends` s`] {` $classfields$ `}` $\leadsto$ `type` c = `class [extends` s`] {` $classfields$ `}`
   ne devait pas être considérée comme un désucrage de l'objet !
   C'est bien d'avoir (ap)pris le corrigé des épreuves précédentes, mais ça ne doit pas se substituer à la réflexion : beaucoup d'étudiants ont répondu que le désucrage pouvait se situer pendant la création de l'AST. Or on ne peut sérieusement envisager le désucrage des constructions objet qu'après avoir effectué la *vérification des types*. Celle-ci est en effet bénéfique à la clarté des messages d'erreurs comme l'on mentionné certains, mais surtout, cette passe est capitale pour obtenir des information indispensables au désucrage, telles que
   – le graphe d'héritage statique (quelle est la surclasse de chaque classe, quelles sont les éventuelles classes filles) ;
   – la nature des accès aux champs d'une variable (notation $x.a$) : le comportement n'est pas le même selon que la variable (x) est un enregistrement ou un objet (à cause de l'héritage de données) ;
   – les affectations d'objets au sens général (initialisations dans une déclaration de variable, affectations manifestes (opérateur `:=`), initialisation des arguments lors d'un appel de fonction) doivent prendre en compte le polymorphisme d'inclusion.
   – etc.
   De façon générale, un désucrage non trivial nécessite de l'information, et se situe souvent après l'analyse sémantique.

   Best-of: Parmi les réponses farfelues, on trouve :
   – Entre le « léopard avec objets » et le « léopard sans objet ».
   – Lors du lexing.

3. Considérons le programme suivant :

```
let
 type C = class
```

```
 {
   var a := 42
   method a_get () : int = self.a
 }
 var c := new C
 var a := 51
in
 c.a := a;
 print_int (c.a_get ());
 print ("\n")
end
```

Proposer une version de ce programme réécrite en Tiger (dépourvue de constructions syntaxiques spécifique à l'objet, donc). À ce stade de l'exercice, on ne demande pas de prendre en compte l'héritage de classe, le polymorphisme d'inclusion, et les méthodes polymorphes.

Correction: Note : il y avait un *bug* dans le sujet initial ; à la ligne 10, au lieu de c.a = a, il aurait fallu lire c.a := a. Le correcteur n'a pas tenu compte des éventuelles erreurs liées à cette coquille, qui de toutes façons n'affectait pas le fond de la question.

```
let
  type C = { a : int }
  function C_new () : C = C { a = 42 }
  function C_a_get (target : C) : int = target.a
  var c := C_new ()
  var a := 51
in
  c.a = a;
  print_int (C_a_get (c));
  print ("\n")
end
```

Il fallait aussi prendre soin à ne pas éclater les attributs des classes, comme de nombreuses copies le proposent : on perd l'encapsulation, et la manipulation d'objets en sera rendu très difficile (par exemple, le passage d'objets en argument ou en retour de fonction, le transtypage vers un type plus général, etc.).

Best-of: Certains ne désucrent pas, il décapent carrément le programme :

```
let
  var a := 51
in
  print_int (a);
  print ("\n")
end
```

voire :

```
print_int (51);
print ("\n")
```

4. On souhaite maintenant traiter le mot-clef `extends` pour ajouter l'héritage de classe. On ne considère ici que l'héritage de *données*, et pas la relation de typage dite de *généralisation* (« est un »), ni l'héritage de comportements (méthodes). Comment procédez-vous ? Appliquez votre proposition au programme ci-dessous.

```
let
  type C = class
  {
    var a := 0
  }
  class D extends C
  {
    var b := 4 + 8 + 15 + 16 + 23 + 42
  }
  var c : C := new C
  var d : D := new D
in
end
```

Correction: Plusieurs solutions : par agrégation, ou par délégation.

Agrégation :

```
let
  type C = { a : int }
  type D = { a : int, b : int }
  function C_new () : C =
    C { a = 0 }
  function D_new () : D =
    D { a = 0, b = 4 + 8 + 15 + 16 + 23 + 42 }
  var c : C := C_new ()
  var d : D := D_new ()
in
end
```

Délégation :

```
let
  type C = { a : int }
  type D = { super : C, b : int }
  function C_new () : C =
    C { a = 0 }
  function D_new () : D =
    D { super = C_new (), b = 4 + 8 + 15 + 16 + 23 + 42 }
  var c : C := C_new ()
  var d : D := D_new ()
in
end
```

Best-of:

– Dans un exemple de désucrage :

```
/* ... */
  print ("push the button Locke!")
/* ... */
```

– Ne pas oublier de relancer le programme toutes les 108 minutes pour ne pas avoir de problème.

5. On s'intéresse ensuite au polymorphisme d'inclusion (en laissant de côté la résolution dynamique de méthodes pour le moment). Dans le programme ci-dessous, quelle(s) opération(s)

nécessite(nt) le polymorphisme d'inclusion, et pourquoi ? (Vous êtes invité à mentionner dans votre réponse la ou les lignes pertinente(s) du listing).

```
1  let
2   type X = class
3   {
4    var t := 42
5   }
6   type Y = class extends X
7   {
8    var u := 51
9   }
10  var x : X := new X
11  var y : X := new Y
12 in
13  x := y;
14  x := new Y
15 end
```

Correction: Il s'agit des lignes lignes 11 et 14, qui font apparaître des affectations polymorphes. Le type de la r-value est un sous-type (différent) de celui de la l-value (*upcast*).

Proposer une solution de désucrage. On ne demande pas de traduire l'intégralité de l'exemple précédent en Tiger ; une argumentation informelle (mais certainement pas vague) sera suffisante.

Correction: Pour prendre en compte les affectation polymorphes, il faut conserver les informations des types dynamiques. Pour chaque type statique, on va donc devoir disposer d'une structure de données nous permettant de stocker un ensemble de données relatives à un type exact, parmi tous les types dynamique possibles compatibles avec le type statique. Cette structure n'est autre qu'un *variant* (ou union généralisée). Or nous ne disposons pas d'une telle structure en Leopard.
Nous allons donc « émuler » cette structure de données, on stockant un couple d'informations :

(a) un numéro identifiant le type exact de l'objet (à l'exécution) ;

(b) les données (attributs) associés à ce type exact, dont le nombre et la nature varient donc en fonction de l'élément précédent !

Le premier membre est un simple entier, tandis que le second est un enregistrement (Leopard ne dispose pas d'unions) dont au plus un champ est non `nil` et contient un enregistrement regroupant les données liées au type exact indiqué par le numéro mentionné ci-avant.
Ceux qui ont lu la version C du Livre d'Appel ont (peut-être) été avantagés, puisqu'il utilise une solution proche de cette proposition, pour représenter l'AST d'un programme dans *Modern Compiler Implementation in C* [1].
À chaque affectation (au sens général, c'est-à-dire : initialisation d'une variable, affectation manifeste, initialisation d'un argument formel d'une fonction), si les types sources et cibles ne sont pas identiques, il faudra insérer une conversion de type. Celle-ci sera chargée de consulter le type exact de l'objet désucré, et de copier les données de celui-ci dans le champ *ad hoc* de l'objet cible.
Un désucrage simplifié (ne faisant aucune mention de `Object` par exemple) est proposé dans l'annexe A.

Attention à ceux qui proposent de désucrer en « découpant » l'objet (*slicing*), c'est-à-dire en ne conservant que les données du type statique ! L'objet doit rester intact en mémoire lors de cette opération, et ne surtout pas perdre de données. Imaginez ce qui arriverait si une `ForExp` de la syntaxe abstraite de Leopard se faisait découper à chaque fois qu'elle était vu comme une `Exp` !

6. Les appels de méthodes polymorphes en Leopard n'utilisent pas des tables de méthodes virtuelles comme en C++. Au lieu de cela, chaque objet comporte une étiquette (un numéro) indiquant quel est son *type dynamique*. Dans ces conditions, quelle approche peut-on employer pour résoudre un appel de méthode polymorphe ?

Correction: La première étape consiste à calculer, pour chaque classe (type abstrait), l'ensemble des types concret recevables (c'est-à-dire, l'ensemble de ses sous-classes). Il faut bien comprendre que cette approche n'est valable que parce que nous somme sous l'hypothèse du monde clos ; on ne s'attend pas à ce qu'un nouveau type vienne sous-classer une classe après que celle-ci ait été compilée.
On transforme ensuite tous les appels de méthodes en trampolines : une fonction chargée de consulter le type exact de l'objet (via son numéro), et un aiguillage (*switch*) déléguant l'exécution effective de la méthode à une fonction spécialisée pour le type exact.

Beaucoup proposent d'utiliser le tag de type dynamique (connu seulement à l'exécution, donc) pour transformer le *nom* de la fonction à appeler (un mécanisme statique) : il y a là une profonde incompatibilité.
Malgré l'énoncé, certains parlent de table de fonctions virtuelles (VFT). On souhaite vraiment une solution sans VFT, cette dernière étant difficile à mettre en œuvre dans un langage comme Leopard (en tout cas, en conservant la solution du désucrage).

7. Proposer une solution de désucrage d'un appel de méthode polymorphe du langage Leopard. C'est l'idée générale de la solution qui est attendue ici, et non la traduction d'un exemple Leopard vers Tiger.

Correction: Tout ou presque a déjà été dit dans la correction de la réponse précédente. Pour chaque méthode virtuelle, nous allons écrire une façade, dont la signature aura un argument supplémentaire par rapport à la méthode initiale : la cible (l'objet) de la méthode. Le type de cet argument doit être le variant polymorphe correspondant au type statique de l'objet, de sorte que l'on puisse lui passer n'importe quel objet désucré ayant le même type statique (mais un type dynamique éventuellement différent). Cette façade effectue le dispatch (à base de `if`, puisque Leopard n'a pas de `switch`) vers l'implémentation correspondant au type dynamique, obtenue en consultant l'étiquette (numéro) stocké dans le variant.
Pour chaque (re)définition de méthode, on crée une fonction avec la même corps, on ajoutant là aussi un argument supplémentaire correspondant à `self`. Ces fonctions seront appelée lors de la sélection sur le type exact évoquée ci-dessus.
Le lecteur intéressé est invité à se reporter à [2] pour plus de détails.

3.3 Vers le Leopard et au delà

1. On souhaite étendre la syntaxe de Leopard pour lui ajouter des constructeurs. Proposer des extensions du langage permettant de définir un constructeur dans une classe, ainsi que d'utiliser celui-ci (les réponses informelles seront tolérées, tant qu'elle sont intelligibles et non ambigües). Donner un exemple illustrant ces ajouts syntaxiques.

Correction: L'ajout d'un constructeur n'a généralement pas posé de problème, mais deux points n'ont pas toujours été bien traités.
Le premier tient à la façon dont sont initialisés les attributs d'une classe dans la spécification initiale de Leopard : ceux-ci prennent leur valeur à l'instanciation d'un objet, mais cette valeur est connue à la définition de la classe ! Or en introduisant des constructeurs, on est en droit d'utiliser des valeurs définie *après* la définition de la classe.
Une solution consiste à modifier la définition des attributs des classes, de sorte à leur retirer leur valeur d'initialisation. En choisissant de réserver le mot `create` pour désigner un constructeur, cela pourrait donner :

```
class A =
{
  /* Attributes. */
  var x_ : int     /* No initialization. */
  var y_ : string  /* Likewise. */

  /* Ctor. */
  create (x : int, y : string) =
    { x_ = x, y_ = y }
}
```

Ici, on a choisi de rendre la spécification du corps du constructeur très simple, en le limitant à une succession d'initialisations (un peu comme dans une création d'enregistrement). Le type-checker aura alors la tâche de vérifier que tous les membres ont bien été initialisés. Voici un autre exemple, où la signature du constructeur est différente, et l'initialisation moins triviale.

```
class A =
{
  var x_ : int
  var y_ : string
  create (x : int) =
    { x_ = x + 1, y_ = chr (x + ord ("0")) }
}
```

Le second point qui nécessitait une certain soin était la création d'objets en utilisant ce constructeur. L'erreur principale à éviter était de considérer le constructeur comme une méthode : en effet, celle-ci aurait nécessité l'existence d'un objet pour pouvoir être invoquée, ce qui aurait donné au mieux une solution bancale, au pire un non-sens. Un syntaxe naturelle est de passer les arguments du constructeur à la suite de l'expression `new` x.

```
/* Create an A. */
var a := new A (42)
```

Attention aux réponses hors-sujet : plusieurs personnes ont parlé de destructeurs, alors que ni le mot ni le concept ne sont évoqués dans l'énoncé !

2. On souhaite désormais pouvoir ajouter plusieurs constructeurs dans une même classe. Que faut-il ajouter au langage pour que ça fonctionne ? Donner des exemples d'utilisations. Quels sont les changements à appliquer au compilateur ? Soyez précis, et indiquez quelles parties sont affectées par cette fonctionnalité.

Correction: Il faudrait que notre compilateur supporte la surcharge de fonctions, et l'appliquer aux constructeurs. Comme dans le cas des fonctions, cette modification nécessite des modifications au niveau de la liaison des noms (on ne peut lier les fonctions/constructeurs uniquement grâce à leur nom) et de la vérification des types (c'est le type-checker qui effectuera les dites liaisons grâce aux types des arguments).

Références

[1] A. W. Appel. *Modern Compiler Implementation in C, Java, ML*. Cambridge University Press, 1998.

[2] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables. the SmallEiffel compiler. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 32 of *Issue 10*, pages 125–141, Athlanta, GA, USA, Oct. 1997.

Annexes

A Proposition de désucrage (simplifié) de l'exemple de l'exercice 3.2, question 5

```
let
  var _id_X := 1
  var _id_Y := 2

  type _contents_X = { t : int }
  type _variant_X = {
    exact_type : int,
    field_X : _contents_X,
    field_Y : _contents_Y
  }

  type _contents_Y = { u : int, t : int }
  type _variant_Y = {
    exact_type : int,
    field_Y : _contents_Y
  }

  function _new_X () : _variant_X =
    let
      var contents := _contents_X { t = 42 }
    in
      _variant_X {
        exact_type = _id_X,
        field_X = contents,
        field_Y = nil
      }
    end

  function _new_Y () : _variant_Y =
    let
      var contents := _contents_Y { u = 51, t = 42 }
    in
      _variant_Y {
        exact_type = _id_Y,
        field_Y = contents
      }
    end

  function _cast_Y_to_X (variant : _variant_Y) : _variant_X =
    _variant_X{
      exact_type = _id_Y,
      field_X = nil,
      field_Y = variant.field_Y
    }

  var x : _variant_X := _new_X ()
  var y : _variant_X := _cast_Y_to_X (_new_Y ())
in
  x := y;
  x := _cast_Y_to_X (_new_Y ())
end
```