Correction du Partiel Compilation : Surcharge

EPITA – Promo 2005 – Documents autorisés

Avril 2003

Correction: Le sujet et une partie de sa correction ont été écrits par Akim Demaille. Benoît Perrot et Raphaël Poss en ont fait la correction, et ont reporté, à la demande expresse d'Akim Demaille, la partie "best-of".

> 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.

Un soucis constant de l'inventeur de langage est la conception de fonctionnalités :

orthogonales qui peuvent s'utiliser librement sans contraintes artificielles dues aux interactions entre fonctionnalités

ouvertes à l'utilisateur la bibliothèque du langage n'a pas plus de droits que l'utilisateur

implémentables garder en tête qu'il faudra un jour le mettre en œuvre dans un compilateur. Dans le cas d'extensions d'un langage, il est souhaitable de plus qu'elles soient

conservatives que les programmes valides du langage souche le restent dans le langage étendu.

Nous allons nous intéresser à certaines formes de *polymorphisme*, interdites au programmeur Tiger, et nous efforcer de l'ouvrir, en tâchant de respecter les critères précédents.

Notes de correction

Les chiffres romains qui peuvent appraître en haut de la copie n'étaient que des marqueurs de correction. Ils ne rentrent pas dans la notation de la copie et ne vous concernent aucunement.

Chaque question est notée de 0 à 4 ; au résultat est affecté un coefficient multiplicatif correspondant au barème.

D'une manière générale, une réponse vide vaut 0.

D'une manière générale, une réponse à 4 correspond à une réponse parfaite.

Les renvois au livre d'Appel sont considérés comme « foutage de gueule », qui auraient pu dans d'autres circonstances entrainer des points négatifs. Ont été considérés comme tels les réponses du style « Cf p. 344 du livre d'Appel » et les schémas en anglais directement recopiés du même livre.

Dans ce qui suit, tout ce qui est entre quotes « » correspondent à des citations brutes de copies.

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.

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 Simula et 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`.

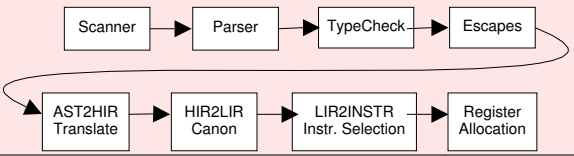
Catégories de réponses	Récompense
La réponse est incongrue. « le dynamic_cast », « polymorphisme de classe », « coercion », « la STL s'appuie sur les conteneurs »	0
La réponse est confuse. « template c'est à dire polymorphisme de classe » « polymorphisme de substitution c'est à dire template »	1
La réponse 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

Quatre formes de polymorphisme viennent d'être exhibées. Nous nous intéresserons aux deux premières, celles que l'on rencontre déjà dans Tiger, mais qui sont interdites à l'utilisateur.

2 Tiger : Remise en Jambes

1. Rappeler quelles sont les grandes phases du compilateur Tiger. On demande un court schéma présentant des boites dont les entrées et sorties sont liées.

Correction:



Catégories de réponses	Récompense
Le schéma se limite à analyse lexicale et syntaxique.	0
Le schéma va jusqu'à l'analyse de type.	1
Le schéma va jusqu'aux représentations intermédiaires.	2
– Le schéma va jusqu'au code assembleur, mais la phase de sélection des instructions est placée après celle d'allocation des registres. – Le schéma est complet, dans le bon ordre mais en anglais (recopiage pur et simple du livre d'Appel)	3
Le schéma est complet et en français.	4

2. Expliquer le déroulement de l'analyse de type des déclarations de fonctions en Tiger.

Correction: Une spécificité de Tiger par rapport à des langages comme ML, ou tout simplement le C etc., c'est que les définitions de type et de fonctions sont "naturellement" récursives : il n'est pas nécessaire d'ajouter du `letrec` et du `and`, ni de faire de prédéclaration. Tout fonctionne par "chunks", ou bloc de déclarations, i.e., une suite non interrompue de déclarations de fonctions (ou de type) est considérée d'un bloc. Ainsi, lorsque le `TypeVisitor` lit le bloc de définition de fonctions qui contient `fact` dans l'exemple 6.1, il place d'abord la signature de celle-ci dans l'environnement, puis il étudie la validité du typage du corps de la fonction, après avoir ajouté les arguments dans l'environnement (ici `n`). Dans le cas de l'exemple 6.2, il place dans l'environnement les signatures de `print_red` et de `print_black`, puis il étudie le corps de `print_red` après avoir inséré l'argument `red` de type `red` dans l'environnement. Une fois le corps de la fonction vérifié, il restaure l'environnement dans son état précédent, et procède de même dans le cas de `print_black`. C'est précisément parce que `print_red` et `print_black` sont toutes les deux dans l'environnement que lors du contrôle du corps de `print_red`, le typage de l'appel à `print_black` ne pose aucun problème.

Ont été plus ou moins acceptées les réponses évoquant :
- le parcours par chunks
- le premier passage pour la signature
- le second passage pour le corps
- les vérifications d'usage sur l'environnement et les différents messages d'erreurs pouvant survenir (la fonction existe déja, etc.)

3. Expliquer le déroulement de l'analyse de type des appels de fonctions en Tiger. Insister sur les différents cas d'erreur possibles.

Correction: On récupère le nom de la fonction appelée, et l'on demande à l'environnement ce qu'il en connaît. Ca peut donner lieu à une première erreur : fonction inconnue. Puis on contrôle le type des tous les arguments en s'assurant que (i) l'arité est bien la même (autant d'arguments formels que d'arguments effectifs), et (ii) que les arguments effectifs sont compatibles avec les arguments formels. "Compatible" ici désigne essentiellement l'égalité (ils doivent être de même type), à l'exception près de `nil` qui peut être un argument effectif pour une place de n'importe quel type `record`. Enfin, cet appel de fonction renvoie comme type celui de la fonction (e.g., un appel `fact` retourne un `int` dans l'exemple 6.1). Ceci peut générer un troisième type d'erreur : le type de retour n'est pas compatible avec celui du contexte. Techniquement, cette erreur n'est pas gérée dans le contrôle de type de l'appel de fonction dans notre architecture de compilateur Tiger ; elle est gérée là où la fonction est appelée.

Ont été acceptées les réponses évoquant :
- la recherche dans l'environnement de l'identifiant de la fonction
- la vérification des types d'entrées, en terme d'équivalence de type et non pas en terme d'égalité de type (`nil` et les record)
- le nombre des paramètres
- la propagation du type de retour
- les différents messages d'erreurs pouvant survenir.

3 OverTiger : Surcharge en Entrée

On souhaite étendre le langage Tiger pour qu'il supporte une forme de surcharge des fonctions comparable à celle du C++. Appelons OverTiger ce langage, dont les primitives sont les mêmes

que celles de Tiger (`print`, `print_int` etc.).

Notes de correction

Une grande majorité rédige les réponses aux trois premières questions en considérant dès 3.1 que le plus gros du travail se fait au type-checking, et qu'il faut expliquer à 3.1 et 3.2 la quantité de travail à rajouter, en plus de la gestion au type-checking, à la syntaxe (« bah en fait il n'y a rien à faire à la syntaxe ») et dans le back-end (« rien non plus à faire puisque le type checking s'en est occupé »).

Best-of:

« // FIXME : some answers were deleted here »

1. Discuter la possibilité d'implémenter la surcharge au sein de l'analyse syntaxique.

Correction: Comme vu auparavant, il faut des informations de typage pour pouvoir résoudre les appels surchargés, par conséquent, à moins de redescendre le contrôle de type dans le parser, c'est inimaginable.

Catégories de réponses	Récompense
– Réponse incongrue. – Réponse du type « il suffit de pouvoir gérer plusieurs déclarations de fonctions de même nom »	0
Réponses du type : – « il n'y a rien à rajouter à l'analyseur syntaxique car la syntaxe n'est pas modifiée par la surcharge » – « il faut typer les appels et déclarations de fonctions au niveau de l'analyse syntaxique » – « c'est difficile car il n'y a pas d'informations de typage à l'analyse syntaxique »	1 à 4

Best-of:

« il faut rajouter un mot-clef `overload` dans la syntaxe »

« il faut calculer les échappements avant »

« il faut rajouter la gestion de '<' et '>' »

« on pourrait le faire mais alors ce serait plus lent et donc moins optimisé »

« Il est quasiment impossible de travailler sur la grammaire concrète. De plus, retoucher à du code bison pour ajouter une fonctionnalité est digne des pirates japonais pendant la 2nde Guerre Mondiale »

2. Discuter la possibilité de gérer la surcharge dans le back-end.

Correction: Super ! Quelle bonne idée ! Être obligé d'implémenter ça pour chaque langage cible ! Sans compter que ça veut dire qu'il faut trouver des représentations intermédiaires (HIR, LIR, ASM) qui supportent la surcharge !
Non, vraiment, c'est pas une bonne idée.

Une très large majorité des copies mentionnent la réponse (nulle) :
« qu'est-ce que le back-end ? »

Réponses typiques	Récompense
– « c'est plus facile dans le back-end car le typage a déjà été réalisé » – « il faut rajouter des instructions de surcharge dans l'assembleur »	0
– « si on attend le back-end pour gérer la surcharge, il y aura des erreurs à l'analyse syntaxique qui a lieu avant » – « implémenter la surcharge dans le back-end apporte de la redondance entre les différents types d'assembleur » – « il faudrait rendre l'analyseur sémantique plus souple pour accepter la surcharge et rajouter des instructions de surcharge au langage intermédiaire »	1 à 4

3. En montrant où sont les différences avec Tiger, en déduire que l'implémentation la plus naturelle de la surcharge se fait pendant l'analyse de type. On expliquera en particulier pourquoi il est inadapté de vouloir résoudre la surcharge après avoir effectué la phase de contrôle de type.

Correction: Dans la question 1.3 on a vu que le choix du code se faisait simplement par examen du type. Il faut donc au moins que le type soit connu, ce qui laisse la possibilité soit de le faire pendant le contrôle de type, soit ensuite. Mais le faire ensuite est mal aisé, en particulier dans le cas où le type de retour dépendrait de la résolution de la surcharge, comme le démontre l'exemple 6.5. Le moment le plus adéquat est donc pendant le contrôle de type.

Catégories de réponses	Récompense
Si les réponses aux questions 3.1 et 3.2 sont justes, les réponses du type « si ça ne peut pas avoir lieu avant ou après, c'est forcément à l'analyse des types ».	1
Les réponses évoquant : – certains types de retour dépendent de la résolution de la surcharge, donc celle-ci doit avoir lieu au plus tard pendant l'analyse des types – pour résoudre la surcharge en entrée, on a besoin des types des arguments, donc la résolution de la surcharge ne peut avoir lieu qu'au plus tôt pendant l'analyse des types	2 à 4

4. Quel message d'erreur pourrait-on vouloir émettre sur le programme `print (51)` ?

Correction: A la façon de G++ dans le cas de la surcharge du C++, plutôt qu'un simple `unknown function`, on aimerait
`unknown function "print (int)"`
`possible candidates: "print (string)"`

Catégories de réponses	Récompense
Aucune réponse ou un message d'erreur d'analyse syntaxique sans surcharge (du type « found ..., *expected* ... »).	0
Réponses du type : « il faudrait faire comme un compilateur C++ ».	1
La réponse indique que le prototype complet de l'appel non résolu doit faire partie du message d'erreur : « unknown function `print(int)` »	2
La réponse indique le prototype non trouvé et la liste des candidats.	4

Best-of:

« si la surcharge est bien implémentée, justement il n'y a pas d'erreur ! »

« l'abus d'alcool est dangereux pour la santé »

5. En déduire l'ensemble des changements à apporter dans le `TypeVisitor` de Tiger pour l'adapter à OverTiger. On n'oubliera pas de parler des objets outils qui servent dans le contrôle de type.

Correction: La mise en œuvre de la surcharge ne concerne que les fonctions, leurs déclarations, et leurs appels. Il faut que chaque appel puisse retrouver la bonne version de la fonction surchargée. Il faudra donc changer le `TypeVisitor` pour que lors de la visite d'une fonction, sa signature en entrée (i.e., les arguments formels) serve de clef, et non plus seulement son nom. Au moment des appels de fonctions, il reste à fabriquer la clé (il suffit pour cela de déterminer le type des arguments), et à interroger l'environnement pour retrouver cette fonction. Il faut donc ajuster l'environnement pour que cette recherche puisse avoir lieu : ou bien les clefs deviennent effectivement la fonction avec sa signature en entrée, ou bien à un nom est associé une liste de signature. Noter que cette dernière approche est plus judicieuse puisse qu'elle permettra de meilleurs messages d'erreur (voir ci-dessus). Le coût du parcours de cette liste peut être maîtrisé en en faisant une table dont la clé est la signature en entrée.

Catégories de réponses	Récompense
Réponses évoquant : – il faut utiliser une liste de listes au lieu d'une liste pour les fonctions – il faut identifier les fonctions par leur types d'arguments en plus de leur nom	1 à 2
Réponses évoquant : – il faut utiliser la signature complète comme clef pour les fonctions dans l'environnement – il faut changer `TypeVisitor` pour utiliser la signature de l'appel lors de la recherche d'une fonction dans l'environnement – il faut modifier l'environnement et utiliser la signature comme clef ou associer une liste de signatures à chaque nom	3 à 4

6. Dans le cas d'un compilateur qui supporte les deux langages Tiger (`tc -x tiger`) *et* OverTiger (`tc -x overtiger`), expliquer comment factoriser le contrôle de type.

Correction: Deux solutions sont possibles : ou bien faire deux visiteurs, ou bien faire un visiteur unique contrôlé par un drapeau. Dans le premier cas, le contrôle de type OverTiger peut donner lieu à un `OverTigerTypeVisitor` qui se contente de spécialiser (i.e., en exploitant le mot clé `virtual`) quelques méthodes de `TypeVisitor`. Bien entendu il reste à s'assurer que l'environnement supporte plusieurs signatures associées à un seul nom. Ou bien, l'environnement est modifié pour associer à chaque nom de fonction un ensemble de signature (`std::set`), et tout le visiteur est adapté à cet ensemble, mais son remplissage est contrôlé par un drapeau, qui déclenche une erreur dès qu'un nom est surchargé en Tiger.

Catégories de réponses	Récompense
La réponse indique qu'il faut avoir deux implémentations de l'analyse de type et qu'on doit pouvoir « choisir » entre les deux.	1 à 2
Réponses évoquant : – on peut implémenter un deuxième `TypeVisitor`, si possible en dérivant le premier – on peut activer la gestion de la surcharge par un drapeau donné en argument au programme	3 à 4

Best-of:

« il suffit d'avoir un #ifdef et de recompiler tc pour changer de comportement. »

« il faudrait faire une surcharge de la fonction `print` avec en paramère un `int` et un autre polymorphe avec en paramètre un 'type'. »

4 AboveTiger : Surcharge en Sortie

On s'intéresse à une extension de OverTiger, AboveTiger, qui prenne aussi en compte la surcharge sur les valeurs de retour des fonctions. Les primitives sont les mêmes que celles de AboveTiger, donc de Tiger (`print`, `print_int` etc.).

1. Montrer que sans aucune restriction, AboveTiger serait un langage mal défini, puisqu'un programme pourrait donner lieu à deux comportements différents.

Correction: Un programme comme le suivant peut afficher `int` ou `string`.

```
let function foo (i: int)    = print ("int")
    function foo (i: string) = print ("string")
    function bar () : int    = 1
    function bar () : string = "1"
in
  foo (bar ())
end
```

Un autre réponse correcte est de citer 6.6 ou 6.7 en mentionnant l'ambiguïté de ce qui est écrit, ou un autre exemple *correct* qui l'illustre aussi.

Best-of:

« il faudrait gérer `public` »

2. Formuler une restriction générale qui interdirait de tels programmes.

Correction: Il semble raisonable d'interdire toute expression qui puisse avoir deux lectures, i.e., exiger qu'un seul typage de toutes les expressions soit valide. Sont admis comme bonnes réponses :
– « il ne faut pas que la résolution aboutisse à une ambiguïté »
– « il ne faut pas pouvoir aboutir à plusieurs comportement possibles »
– « il faut exiger une seule solution à l'équation des types »

Best-of:

Réponse majoritaire ... et nulle « il faut interdire la surcharge en sortie en même temps que la surcharge en entrée. »

Second plus grand score ... tout aussi nul « il faut interdire la surcharge en sortie quand il n'y a pas d'arguments. »

...Nul aussi, mais heureusement, plus rare « on adopte un comportement par défaut (?) en définissant une syntaxe spéciale pour la surcharge »

« il suffit d'utiliser toujours la première déclaration »

3. Montrer que cette extension, à cause de la restriction que l'on vient d'introduire, n'est pas conservative.

   Correction: Le fait d'introduire la surcharge modifie le masquage des fonctions. Le fait d'introduire la restriction dans la question précédente conduit au rejet de certains programmes Tiger, comme le 6.3.

4. Montrer comment mettre en œuvre cette restriction en expliquant le contrôle de type des appels de fonctions.

   Correction: L'exemple 6.7 montre particulièrement bien qu'il faut d'abord trouver, parmi les signatures possibles de la fonction appelée (`foo`), laquelle est la seule possible. Pour ce faire, il nous faut demander aux arguments effectifs quels peuvent être leurs types, au pluriel, puisque par exemple l'appel à `ab` en premier argument de `foo` peut avoir pour type `a` ou `b`. Lorsque les types possibles des arguments et les types possibles de la fontion appelée sont croisés, il ne doit rester qu'un seul choix. Une fois cette signature déterminée, il faut repartir dans les arguments, de façon à en faire le véritable contrôle de type. Par exemple, on part à nouveau dans l'appel à `ab`, cette fois-ci en lui précisant que son type de retour doit être `a`.
   Il est très clair que le visiteur devra être beaucoup plus sévéremment modifié, puisqu'il devra permettre une interrogation sur les types possibles d'une expression, et spécialement dans le cas d'un appel de fonction.

5 BeyondTiger

Si jamais vous estimez ne pas avoir eu suffisamment de place pour démontrer vos facultés de concepteur de langage, vous êtes invité(e) à décrire ici comment vous envisagez d'aller encore plus loin dans la surcharge qu'AboveTiger.

Correction: On pourrait par exemple rajouter de la surcharge sur les variables.
On pourrait vouloir se donner une syntaxe pour désambiguïser par une annotation des types plutôt que d'obliger à introduire une variable intermédiaire. Par exemple, le programme suivant n'est pas du OverTiger valide :

```
let type a = {}
    function print (a : a) = print ("a\n")
    type b = {}
    function print (b : b) = print ("b\n")
in
  print (nil)
end
```

on est obligé d'écrire :

```
let type a = {}
    function print (a : a) = print ("a\n")
    type b = {}
    function print (b : b) = print ("b\n")
    var nil_a : a := nil
in
  print (nil_a)
end
```

On préférerait plutôt :

```
let type a = {}
    function print (a : a) = print ("a\n")
    type b = {}
    function print (b : b) = print ("b\n")
in
  print (nil :: a)
end
```

Mais ce n'est pas réellement lié à la surcharge. D'ailleurs, Tiger simple pourrait bénéficier de cette extension dans le cas suivant :

```
let type a = {}
    var  a := nil :: a
/* ... */
```

Best-of:

« il faut implémenter `const` après la surcharge »

« on pourrait faire du typage dynamique ; mais c'est _mal_ (Java), ça plante à l'exécution »

« mon extension s'appellerait Lion, avec de la surcharge d'opérateurs »

6 Exemples de Programmes

Cette section a double vocation : présenter quelques exemples de programmes Tiger qui pourraient réveiller les souvenirs de ceux qui auraient peu travaillé sur `tc`, mais aussi présenter des exemples des extensions étudiées pendant cet examen. Vous êtes chaudement encouragés à faire référence à ces exemples dans votre copie.

6.1 Tiger : Factorielle

```
let function fact (n : int) : int =
    if n then n * fact (n - 1) else 1
in
  fact (51)
end
```

6.2 Tiger : R'n B (Beurk)

```
let type red   = {h : int,    t : black}
    type black = {h : string, t : red}
    function print_red (red~: red) =
      if red != nil then
        (print_int (red.h); print_black (red.t))
    function print_black (black : black) =
      if black != nil then
        (print (black.h); print_red (black.t))
in
  print_red (red {h = 123, t = black {h = "456", t = nil}})
end
```

6.3 Tiger : Masquage

Masquage des primitives `print` et `chr`.

```
let function print (i : int) = print_int (i)
    function chr   (s : string) : string = s
in
  print (chr ("a"))
end
```

6.4 OverTiger : R'n B (OverBeurk)

```
let function print (i : int) = print_int (i)
    type red   = {h : int,    t : black}
    type black = {h : string, t : red}
    function print (red : red) =
      if red != nil then
        (print (red.h); print (red.t))
    function print (black : black) =
      if black != nil then
        (print (black.h); print (black.t))
in
  print (red {h = 123, t = black {h = "456", t = nil}})
end
```

6.5 OverTiger : Id

```
let function id (s : string) : string = s
    function id (i : int)    : int    = i
in
  (id ("1") = "1") = (id (1) = 1)
end
```

6.6 AboveTiger : Read

En admettant l'existence des primitives `read_string` qui lise l'entrée standard et retourne une chaîne de caractère, et `read_int` qui fasse de même pour un nombre.

```
let function read () : int    = read_int ()
    function read () : string = read_string ()
in
  print_int (read ());
  print     (read ())
end
```

6.7 AboveTiger : AB

```
let type a = {}
    type b = {}
    function ab () : a = a {}
    function ab () : b = b {}
    function  b () : b = b {}
    function foo (x : a, y : a) = print ("aa\n")
    function foo (x : a, y : b) = print ("ab\n")
    function foo (x : b, y : a) = print ("ba\n")
in
  foo (ab (), b ())
end
```