Correction du Partiel CMP1-TYLA

EPITA – Promo 2010
Tous documents (notes de cours, polycopiés, livres) autorisés

Janvier 2008 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain et Akim Demaille. Le *best of* est tiré des copies des étudiants.

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.

Cette épreuve est longue, le but est d'en faire un maximum, le mieux possible. Avant toute chose, une lecture complète du sujet est recommandée.

1 Incontournables

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

1. La surcharge de fonctions en C++ est un mécanisme qui dépend des types à l'exécution. Vrai ou faux ?

   Correction: C'est faux : c'est un mécanisme statique, qui dépend des types *statique* des arguments effectifs, résolu à la compilation.

2. L'utilisation de fonctions virtuelles en C++ est incompatible avec la compilation séparée. Vrai ou faux ?

   Correction: Faux. Le modèle de compilation du C++ permet la compilation séparée. L'implémentation des méthodes polymorphes repose habituellement sur un mécanisme de *tables de fonctions virtuelles*, et extensible sans intrusion.

3. Parmi ces propositions, laquelle (lesquelles) est (sont) un pré-requis pour le typage statique fort dans un langage orienté objet ?
   (a) une liaison des noms résolue à la compilation ;
   (b) l'absence d'instructions de transtypage (*cast*) ;
   (c) l'absence de méthodes polymorphes abstraites (appelées fonctions membres virtuelles pures en C++) ;
   (d) la présence automatique d'une surclasse au sommet de toute hiérarchie de classes (`Object`, par exemple).

   Correction: Seule les réponses 3a et 3b sont valides.
   – 3a est bien sûr un pré-requis fort : sans liaison des noms statique, pas de typage statique fort ;
   – avec les casts, on introduit une une faille dans le système de typage ;
   – la présence de méthodes polymorphes abstraites ne pose aucun problème du point de vue du typage : dit grossièrement, le polymorphisme d'inclusion garantit que les objets manipulés fournissent les interfaces attendues ;
   – 3d est indépendant de la sûreté du typage : on pourrait par exemple s'en passer dans Tiger.

4. Quelle est le type (de Chomsky) du langage engendré par

$$S \to X|Y \qquad X \to Zp \qquad Y \to o \qquad Z \to pS$$

Correction: Ce langage est $p^n o p^n, n \in \mathbb{N}$ bien connu pour être hors-contexte (type 2), et non rationnel.

Best-of:
- Le type du langage est anagramme.
- Non réguliaire.
- En remplaçant, on obtient $S = p \dots p = p^n \cdot p^n = p^{2n}$ avec $n \in \mathbb{N}$ ou $S = 0$, mais après, à part dire que ce n'est pas du parenthèsage, je ne vois vraiment pas.
- Le langage engendré par la grammaire [. . .] est infini.
- p^{2n}
- $p^{2n} o^{2n}$ est le langage engendré.
- $p^2 o$
- Le langage engendré est de type 3 : p^n
- Le langage est $(pp)^*$, c'est donc un langage rationnel.

. . . De plus, on peut représenter cette grammaire par l'expression rationnelle $L = p^n o p^n, n \geq 0$. Donc ce langage est rationnel.
- Au moins de type 0.
- $p^n op$ ou o

2 Let's Target C++ !

On se propose dans cette partie d'ajouter un *back-end* à notre compilateur Tiger ciblant le langage C++. Nous souhaitons donc pouvoir via cette nouvelle cible traduire un programme Tiger en un programme C++ équivalent (c'est-à-dire, dont les comportements sont similaires).

Notez que cette approche est devenue classique ; de nombreux compilateurs ciblent un autre langage, qui traduit ensuite le programme vers un langage d'assemblage (par exemple). On peut citer SmartEiffel (qui cible le C et le *bytecode* Java), GHC (compilateur Haskell, qui cible le C), le compilateur Stratego (qui cible le C), etc.

1. Pour quelles raisons peut-on souhaiter bénéficier de ce nouveau back-end ? Autrement dit, à quoi cela peut-il servir ? Citez des cas d'utilisation possibles.

Correction:
- Cela permet de s'affranchir de l'écriture d'un back-end ! Cette tâche serait dévolue au compilateur C++.
- En ciblant C++, on s'offre tout l'environnement qui vient avec le langage : bibliothèques C et C++, debugger, etc.
- Il existe des compilateurs C++ ciblant la majorité des plates-formes matérielles existantes : cette transformation permettrait donc d'étendre les cibles de notre compilateur Tiger en s'appuyant sur le compilateur C++.

Best-of:
- Je pense que ça ne sert à rien.
- La principale raison est de nous faire bosser, ensuite d'avoir un traducteur Tiger-C++ automatique.
- Le C++ offre un grand nombre de ~~Library~~ Librairies [. . .].
- Le back-end est très utile pour la détection d'un contexte.
- Cela permet de faire genre « J'ai un bête de back-end ».

2. Citer les 5 phases d'une partie frontale (*front-end*) d'un compilateur comme `tc`.

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

Best-of:
- Allocation des registres.
- [. . .] bibliothèque, prototypes des fonctions.
- Scanning : les sources sont décapées en tokens. [. . .]
- Traduction du code en langage assembleur (bytecode)
- Création de l'exécutable
- Preprocessing, lexing, parsing. . . What else?

3. Où situer la phase de traduction vers le C++ dans le compilateur ? Justifiez votre réponse.

Correction: On pourrait penser qu'il est possible produire une traduction Tiger vers C++ dès que l'AST est construit. Il n'en est rien. Certains traits de Tiger nécessitent un peu d'analyse sémantique pour obtenir l'information nécessaire à une traduction en C++. Par exemple, les déclarations de variables Tiger permettent une forme limitée d'inférence de type (le type est alors calculé à partir de la valeur d'initialisation). Ce comportement n'est pas possible en C++ : tous les types doivent être manifestes. On ne peut donc pas traduire avant la phase de vérification des types.
Par ailleurs, et sans être une condition nécessaire, une manipulation de programme non triviale gagne toujours a être placée après l'analyse sémantique, de sorte que les erreurs du fichier d'entrée soient attrapées par le front-end du compilateur Tiger, et non par le compilateur C++ lorsqu'il traite le fichier C++ généré.

Best-of:
- Au moment de l'exécution de l'AST.
- La phase de traduction sera dans le générateur de parseur de grammaire, chaque ligne qui va être dépilé sera traduite en C++.
- Avant transtypage et grammaire, car ils sont différents selon que ce soit en C++ ou en Tiger.
- À la fin (ô_o").

4. Y'a-t-il des modifications à apporter aux phases existantes du front-end ? Si oui, citez-les ; si non, expliquez pourquoi.

Correction: Non : cette traduction se situe en aval du front-end.

Best-of:
- Non, car les phases existantes poutrent déjà bien assez.
- Non, ce n'est pas nécessaire grâce aux tables de symboles.
- Il faudra modifier la vérification de type des paramètres car le C++ permet la surcharge paramétrique avec le mot clé « template ».
- Il y a toujours des modifications à apporter :/
- Oui sûrement. . .
- Il faut rajouter une nouvelle phase d'analyse lexicale ainsi qu'une nouvelle phase d'analyse syntaxique qui cette fois-ci concernera le code C++ obtenu aux phases précédentes.

5. Le langage Tiger permet de définir des variables locales via la construction `let`. Comment peut-on gérer cela lors de la traduction en C++ ?

Correction: Il suffit de se rappeler que la syntaxe `let ... in ... end` est une forme de sucre. En notant t_e le type de l'expression e (différent de `void`), on peut utiliser la règle de réécriture suivante :

let var $x : t_x := v$ **in** e **end**
$\rightsquigarrow$ **let function** $f_e\ (x : t_x) : t_e = e$ **in** $f_e\ (v)$ **end**

qui se traduit en C++ :

$t_e\ f_e\ (t_x\ x)\ \{\ e;\ \}$ *// Function encapsulating* e.
int main () { $f_e\ (v)$; }

Par ailleurs, on peut aussi utiliser les accolades pour limiter la portée d'une variable en C++. Enfin, on peut complètement faire abstraction de ce problème en utilisant la fonctionnalité de renommage de tous les identifiants en instances uniques du compilateur : la gestion des portées deviendrait alors inutile.

Best-of:
- Dans certains cas, passer les variables en globales est nécessaires, mais bon, ça reste sale.
- En utilisant des variables imbriquées.
- À l'aide d'une union.
- On peut gérer cela en permettant la surcharge dans les déclarations.
- On peut les gérer avec la macro `#define`.
- En définissant un nouveau type polymorphe.

6. Les fonctions imbriquées n'existent pas en ISO C++ ; comment peut-on les traduire ?

Correction: Une solution possible : « Désimbriquer » les fonctions, et passer aux fonctions internes des arguments supplémentaires correspondant aux variables non locales qu'elles utilisent. Cette opération peut se faire directement en tant que transformation de programme de Tiger vers Tiger. Par exemple, le programme suivant

```
let function f () : int =
    let var a := 42
        function g () : int = a
    in g () end
in f () end
```

pourrait être traduit en

```
let function f () : int =
    let var a := 42
    in g (a) end
   function g (a : int) : int = a
in f () end
```

Certains on proposé la mise en ligne (*inlining*) comme solution : il s'agit d'une réponse partielle, car elle ne permet pas de traiter les fonctions récursives.

Best-of:
- Grâce à des fonctions récursives.
- En faisant une double récursion.
- On peut utiliser des `if` et des `else` à la place.
- Grâce à des fonctions virtuelles.

7. Le mot clef `return` n'existe pas en Tiger, et il va pourtant falloir le faire apparaître dans le code C++ produit. Quelle règle (algorithme) faut-il suivre pour ce faire lors de la traduction du code Tiger vers C++ ?

Correction: `return` apparaît dans un corps de fonction retournant une valeur : on ne va donc considérer que ce cas-là. On rappelle que le corps d'une fonction est une expression (ast::Exp) ; c'est cette expression qui doit être retournée, le cas échéant. Le plus simple est probablement de procéder en deux temps :

(a) appliquer la *règle de réécriture* (transformation de programme) suivante :

```
function f (args) : t = e
⇝ function f (args) : t =
    /* New body of the function. */
    let _result : t = e in _result end
```

(b) puis, au niveau du traducteur en code C++, identifier le motif correspondant au nouveau corps de cette fonction réécrite pour traduire l'ensemble ainsi :

```
t function (args) { t _result = e; return _result; }
```

Best-of:
- La coloration syntaxique.
- Il faut le définir dans trigger comme expression.
- Pour faire suivre le résultat nous devons implémenter une méthode d'affichage du résultat sur la sortie standard.
- [. . .] Puis placer l'équivalent de cette expression entre les parenthèses du `return`.

8. Intéressons-nous aux types. Pour chaque morceau de code Tiger ci-dessous, donner une proposition de traduction C++ (répondre sur votre copie) :

```
type t = int
var t := 42
```

Correction: Tiger définit cinq espaces de noms « cloisonnés » pour les identifiants (trois dans la version sans objet, dite « Panther ». En C++, on ne peut pas aussi facilement jouer avec les genres (*kinds*) des symboles présents dans une même portée, et en particulier avoir un `typedef` et une variable portant le même nom dans une même portée. Il faut donc faire porter l'information de genre sur le nom du symbole pour que le compilateur tolère la traduction.

```
typedef int type_t;
int var_t = 42;
```

```
var string := "Je clair, Luc, ne pas ?"
```

Correction: Aucune difficulté réelle ici : il semble plus sain et plus simple de choisir d'utiliser la classe standard des chaînes de caractères du C++ pour traduire celles de Tiger. Dans ce cas, l'en-tête standard `string` doit être introduit en amont pour que std::string soit connu.

```
#include <string>
/* ... */
std::string var_string =  "Je clair, Luc, ne pas ?";
```

```
type names_t = array of string
```

Correction: On utilise ici aussi un type de la bibliothèque standard du C++ (`vector`) pour traduire le type `array` de Tiger.

```
#include <vector>
#include <string>
/* ... */
typedef std::vector<std::string> type_names_t;
```

```
type point2d = { row : int, col : int }
var p := point2d { row = 3, col = 14 }
```

Correction: La traduction d'un enregistrement Tiger vers une `struct` C++ est naturelle. Pour conserver la même sémantique que Tiger, ces structures sont manipulées par pointeurs (ce qui devrait nécessiter une gestion mémoire ad hoc, inexistante dans la traduction ci-dessous).

```
struct type_point2d
{
  type_point2d (int row, int col) : row (row), col (col) {}
  int row;
  int col;
};
type_point2d* p = new type_point2d (3, 14);
```

```
type int_list = { head : int, tail : int_list }
var l := int_list { head = 51, tail =
       int_list { head = 2501, tail =
         int_list { head = 2097, tail = nil }}}
```

Correction: Peu de nouveautés par rapport au précédent exercice de traduction, si ce n'est la traduction du `nil` de Tiger en un pointeur nul C++, qui confirme la nécessité de manipuler nos traductions d'enregistrement via des pointeurs.

```
struct type_int_list
{
  type_int_list (int head, type_int_list* tail)
    : head (head), tail (tail) {}
  int head;
  type_int_list* tail;
};
type_int_list* l =
  new type_int_list (51,
    new type_int_list (2502,
      new type_int_list (2097, 0)));
```

```
type list1 = { head : int, tail : list2 }
type list2 = { head : int, tail : list1 }
```

Correction: Dans ce dernier exemple, la difficulté tient au fait que le C++ ne sait utiliser `list2` à l'intérieur de `list1` avant d'avoir vu la définition du premier, ou au moins une pré-déclaration. D'où l'insertion d'une telle pré-déclaration. Le traducteur peut ici être plus ou moins malin : par simplicité, il pourrait systématiquement introduire des *forward declarations*, ou bien n'émettre que celle qui sont indispensable à la résolution des noms d'entités mutuellement dépendantes.

Correction:

```
// Forward declaration.
struct type_list2;

struct type_list1
{
  type_list1 (int head, type_list2* tail)
    : head (head), tail (tail) {}
  int head;
  type_list2* tail;
};

struct type_list2
{
  type_list2 (int head, type_list1* tail)
    : head (head), tail (tail) {}
  int head;
  type_list1* tail;
};
```

9. Parmi les propositions que vous avez écrites en réponse à la question précédente, en est-il qui pose(nt) des problèmes du point de vue de la gestion de la mémoire (allocation et libération) ?

Correction: Les entités composites, comme les enregistrements (mais aussi les objets et éventuellement les tableaux) sont allouées dynamiquement : il faudrait logiquement un mécanisme de libération automatique et transparent vis-à-vis de la traduction.

Best-of:
- Non, car tout objet créé est détruit par son constructeur.
- Non, car en C++, la mémoire est gérer par le langage.

10. En Tiger, les opérateurs de comparaison (=, <>, <, >, <=, >=) sont surchargés pour tout ou partie des types du langage. Comment faut-il les traduire en C++ ?

Correction: Traitons séparément le cas des opérateurs d'égalité/non-égalité (=, <>) et celui des opérateurs d'inégalité (<, >, <=, >=).
- Pour les types atomiques (`int` et `string`), la comparaison s'effectue *par valeur* : si on a pris soin d'utiliser la classe std::string pour traduire le type Tiger `string`, on peut utiliser les opérateurs C++ == et != pour comparer des entiers et des chaînes de caractères. Pour les types composites (enregistrements, tableaux et objets), la comparaison se fait *par adresse* : là encore, si on a pris soin de traduire ces objets par des pointeurs, on pourra utiliser en C++ les opérateurs homologues à ceux de Tiger.
- Les inégalités ne peuvent avoir lieu qu'entre couples d'entiers et couples de chaînes de caractères. Avec les même hypothèses que dans la première partie du point précédent, on peut traduire en utilisant les même opérateurs d'inégalité en C++ (c'est immédiat pour `int`, et la bibliothèque standard du C++ définit les opérateurs ad hoc pour std::string).

Best-of:
- La surcharge existe aussi en C++. Il suffit d'utiliser des templates.
- Il faut templater le code C++.
- Utiliser `sizeof`.

11. Traduisez le code Tiger ci-dessous en C++. (*Hint* : il est probablement sage de faire un essai sur un brouillon au préalable.)

```
let
  type bool = int
  var true  : bool := 1
  var false : bool := 2

  /* Of course, these won't work on negative integers,
     but we don't care. */
  function is_even (x : int) : bool =
    if x = 0 then true else is_odd (x − 1)
  function is_odd (x : int) : bool =
    if x = 1 then true else is_even (x − 1)
in
  if is_even (3) then
    is_odd (51) /* Yes, and no. */
  else
    42
end
```

Correction:

Note 1 Il y avait une typo dans l'énoncé initial, où `false` valait 2 au lieu de 0, mais ça ne change rien à l'exercice.

Note 2 Il y avait même plus d'une typo, car le code à traduire était incomplet ! Mais encore une fois, cela n'avait aucune incidence sur l'exercice. Merci à ceux qui ont signalé cette erreur ! Le code correct aurait été :

```
let
  type bool = int
  var true  : bool := 1
  var false : bool := 2

  /* Of course, these won't work on negative integers,
     but we don't care. */
  function is_even (x : int) : bool =
    if x < 0 then false
    else if x = 0 then true else is_odd (x − 1)
  function is_odd (x : int) : bool =
    if x < 0 then false
    else if x = 1 then true else is_even (x − 1)
in
  if is_even (3) then
    is_odd (51) /* Yes, and no. */
  else
    42
end
```

Correction: Les difficultés ici étaient :
– le type `bool` qui ne peut être redéfini en C++, de même que `true` et `false` ;
– les fonctions mutuellement récursives qui nécessitent au moins une pré-déclaration (*forward declaration*) ;
– la structure `if` fonctionnelle, à traduire correctement.

```
typedef int type_bool;
type_bool var_true = 1;
type_bool var_false = 0;

// Forward declarations.
type_bool is_even (int x);
type_bool is_odd (int x);

type_bool is_even (int x)
{
  return (x == 0) ? var_true : is_odd (x - 1);
}

type_bool is_odd (int x)
{
  return (x == 1) ? var_true : is_even (x - 1);
}

int main ()
{
  is_even (3) ? is_odd (51) : 42;
}
```

12. Même exercice.

```
let
  type Exp = class {
    /* abstract */ method eval () : int = 0 /* Dummy value. */
  }
  type Num = class extends Exp {
    var val := 0
    method eval () : int = self.val
  }
  type Bin = class extends Exp {
    var lhs : Exp := new Num
    var rhs : Exp := new Num
    method eval () : int = self.rhs.eval () + self.lhs.eval ()
}
  var b1 := new Num
  var b2 := new Num
  var b := new Bin
in
  b1.val := 42;
  b2.val := 51;
  b.lhs := b1;
  b.rhs := b2;
  print_int (b.eval ())
end
```

Correction: Note : Il y avait encore une typo ici : les attributs `lhs` et `rhs` de `Bin` auraient dû être de type statique `Exp`, alors que le sujet donné en examen leur attribuait (implicitement) le type statique `Num`. (Je n'ai bien entendu pénalisé personne sur ce point.)
Il n'y a pas de grosse difficulté dans cette traduction. Il faut essentiellement se rappeler que :
– les objets se manipulent par pointeurs ;
– toutes les méthodes sont virtuelles *et* concrètes ;
– les initialisations de membres d'objets se font avec des constructeurs.
Le problème des builtins n'a pas été évoqué jusqu'à présent. Ici, on a choisi de traduire l'appel à `print_int` en appel à std::printf, mais on aurait tout aussi bien pu reposer sur une routine C++ `print_int` fournie par un *runtime* accolé à la traduction.

```
#include <cstdio>

struct Exp
{
  virtual int eval () { return 0; }
};

struct Num : Exp
{
  int val;
  Num () : val (0) {}
  virtual int eval () { return this->val; }
};

struct Bin : Exp
{
  Num* lhs;
  Num* rhs;
  Bin () : lhs (new Num), rhs (new Num) {}
  virtual int eval () { return this->lhs->eval () +
                              this->rhs->eval (); }
};

Num* b1 = new Num;
Num* b2 = new Num;
Bin* b = new Bin;

int main ()
{
  b1->val = 42;
  b2->val = 51;
  b->lhs = b1;
  b->rhs = b2;
  std::printf ("%d", b->eval ());
}
```

13. **Question Banco** Après avoir traduit un programme Tiger vers son équivalent C++, puis compilé ce dernier pour un microprocesseur de l'architecture Intel 64, on souhaite l'exécuter dans un debugger comme GDB. Cependant, les messages d'erreur font référence au code C++ généré, et pas au code Tiger originel. Comment peut-on améliorer cela ?

Correction: On peut faire en sorte que notre générateur de code C++ produise des directives `#line`, faisant référence aux localisations du fichier Tiger original.

Best-of:
– [. . .] Ou être une GDB-star et débugger en AMD64.

14. **Question Super Banco** On souhaite pouvoir utiliser une bibliothèque C++ existante depuis un programme Tiger traduit vers C++ grâce à notre nouveau back-end. Que faut-il ajouter à notre compilateur ?

Correction:
– La liaison et le typage des fonctions s'effectue à la compilation en Tiger : il faut donc annoncer les signatures des routines C++ provenant des bibliothèques à utiliser. On peut pour cela utiliser la construction `primitive` de Tiger.
– Malgré cette précaution, les types C++ ne sont pas nécessairement connus de Tiger (par exemple `long`). Il faut donc les traduire. Plusieurs solutions sont possibles : implémenter les correspondances/conversions de types nécessaires dans `tc`, présenter une interface C++ adaptée en créant des fonctions (C++) « enveloppant » les fonctions de la bibliothèque, etc.
– Les prototypes des fonctions de la bibliothèque utilisées doivent être annoncées au compilateur C++ également : il faut donc injecter les directive `#include` nécessaires.
– Le *pipeline* de `tc` n'inclut pas de réelle édition de liens : la bibliothèque Tiger standard est soit injectée dans le code émis, soit fournie à l'exécution. Si l'on décide de confier à `tc` la tâche de faire compiler et lier le code C++ émis, il sera nécessaire d'inclure la (les) bibliothèques C++ externes utilisées.
– Enfin, il reste divers problèmes de noms qu'il faut ajuster : traduire le `_main` en `main` et ajuster sa signature, indiquer que `string` provient de l'espace de noms `std`, etc.

3 The T-Files

En guise de dessert, nous vous proposons un petit exercice consistant à augmenter les fonctionnalités des entrées-sorties (I/O) en Tiger. Le langage actuel, tel que défini dans le Manuel de Référence du Compilateur Tiger, est assez pauvre en routines d'entrées-sorties, et ne comporte aucune abstraction (cf. annexe A, page 15).

On décide donc d'augmenter le prélude du langage avec les déclarations suivantes [1] :

```
/* The type of a stream. */
type stream

/* The standard streams. */
var stdin : stream
var stdout : stream
var stderr : stream

/* Open file with name PATH and obtain a stream. MODE can be "r", "w"
  or "a" for reading, writing or appending. The file will be created
  if it doesn't exist when opened for writing or appending; it will
  be truncated when opened for writing. Add a 'b' to the mode for
  binary files. Add a "+" to the mode to allow simultaneous reading
  and writing. If the file cannot be open or the mode is ill-formed,
  raise a runtime error: "fopen: invalid path." or "fopen: invalid
  mode." */
primitive fopen (path : string, mode : string) : stream
/* Close stream STREAM. If the file handle is invalid, raise a
  runtime error: "fopen: invalid stream." */
primitive fclose (stream : stream)

/* Print STRING on STREAM. */
primitive fprint (string: string, stream : stream)
/* Output INT in its decimal canonical form (equivalent to "%d" for
  fprintf()). */
primitive fprint_int (int: int, stream : stream)
/* Flush the buffer of STREAM. */
primitive fflush (stream : stream)
/* Read a character from STREAM. Return an empty string on an end of
  file. */
primitive fgetchar (stream : stream) : string
```

Étant donné que seul le front-end de `tc` a été vu en cours pour le moment, les réponses non formelles sont acceptées. L'objectif est de réfléchir et de comprendre quels sont les impacts de cette extension.

1. Que faut-il changer dans le front-end de `tc` (i.e., les étapes jusqu'à la représentation intermédiaire dans le langage Tree) ?

1. Les plus attentifs auront remarqué que le code de cette extension du prélude n'est pas du Tiger valide : le langage ne permet pas de déclarer des types et des variables sans leur donner une valeur. Ici, c'est le runtime qui fournira les implémentations absentes. Nous considérerons pour cet exercice que cette syntaxe est valide.

Correction: La seule nouveauté du point de vue du front-end est le type `stream`. Celui devra faire l'objet d'un traitement particulier lors de la vérification des types. Concrètement, cela se traduira par
– l'ajout d'un type « type::Stream » à la hiérarchie des types de Tiger ;
– la prise en compte de ce type dans type::TypeChecker.
Ce nouveau type affecte également la traduction vers la représentation intermédiaire : une variable de type `stream` doit être traduite en un arbre Tree. On peut par exemple choisir une traduction simple et naïve consistant à confondre un flux et le descripteur de fichier associé. Un objet de type stream se verrait donc traduit par une valeur numérique entière.
(La spécificité syntaxique de la déclaration « **type** stream » n'était pas à prendre en compte, donc nous ne traiterons pas ce point.)

Best-of:
– Après le typage [il] faudra créer une étape qui devra fermer tous les fichiers qui n'ont pas été fermer.

2. De même, faut-il enrichir la définition du langage Tree, utilisé pour la représentation intermédiaire par `tc` ? Si oui, quels sont les changements nécessaires ? Si non, pourquoi ?

   Correction: Non : l'interface du runtime vient masquer les nouveautés introduites par cette extension. Cela n'exige aucune modification de Tree.

3. À votre avis, faut-il apporter des changements dans le back-end de `tc` (la partie qui va de la représentation intermédiaire au langage d'assemblage, par exemple MIPS) ? En particulier, est-ce que ces parties du back-end seront affectées :
   – l'allocation de registres ;
   – le runtime Tiger (support du langage à l'exécution) ;
   – l'émission de code en langage d'assemblage.

   Correction: Seul le runtime est affecté : cela signifie qu'il devra fournir les routines ad hoc, ainsi que les « objets » correspondant aux flux standard (`stdin`, `stdout`, `stderr`). Les autres parties du back-end sont orthogonales à cette extension.

   Best-of:
   – Allocation de registres : oui.
   [. . .]
   Émission de code assembler : oui.
   Car l'allocation de registres servira à stocké les fichier ouvert, et l'émission en code assembleur pour préciser que des fichiers ont été ouvert.
   – Il faudra des registres pour les différents flux.

4. **Question Mega Banco** Cette extension est plutôt simpliste. En particulier
   – elle est assez pauvre en routines ;
   – la gestion d'erreurs est brutale (sortie avec une erreur de runtime) ;
   – les flux d'entrées et de sorties ne sont pas distingués (il n'y a qu'un seul type `stream`).
   Proposez des idées pour améliorer la proposition initiale.

   Correction:
   – Définir plusieurs types `stream` (entrée, sortie, entrée/sortie).
   – Encapsuler données et routines traitant des flux dans une classe.
   – Stocker plus d'information dans un `stream`, comme son état courant (ouvert/-fermé, en fin de fichier, dernière erreur survenue, etc.).
   – Une vraie gestion des erreurs.
   – Une interface plus riche : `fseek`, `fprintf`, `fscanf`, possibilité de sérialiser/désérialiser des objets Tiger.

Best-of:
– Chuck Noris l'a fait.

4 À propos de ce cours

Je copie ~~un peu~~ complètement sur Akim, et procède à un petit debriefing discrétisé sur quatre questions.

Bien entendu je m'engage à ne pas tenir compte des renseignements ci-dessous pour noter votre copie. Ils ne sont pas anonymes, car je suis curieux de confronter vos réponses à votre note. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Vous pouvez cocher plusieurs réponses par question. Répondez sur les feuilles de QCM qui vous sont remises.

1. Travail personnel
   - **a** Rien
   - **b** Bachotage récent
   - **c** Relu les notes entre chaque cours
   - **d** Fait les annales
   - **e** Lu d'autres sources
2. Ce cours
   - **a** Est incompréhensible et j'ai rapidement abandonné
   - **b** Est difficile à suivre mais j'essaie
   - **c** Est facile à suivre une fois qu'on a compris le truc
   - **d** Est trop élémentaire
3. Ce cours
   - **a** Ne m'a donné aucune satisfaction
   - **b** N'a aucun intérêt dans ma formation
   - **c** Est une agréable curiosité
   - **d** Est nécessaire mais pas intéressant
   - **e** Je le recommande
4. L'enseignant
   - **a** N'est pas pédagogue
   - **b** Parle à des étudiants qui sont au dessus de mon niveau
   - **c** Me parle
   - **d** Se répète vraiment trop
   - **e** Se contente de trop simple et devrait pousser le niveau vers le haut

Bonne année et meilleurs vœux !

Annexes

A Bibliothèque Standard du langage Tiger

Le listing ci-dessous contient les déclarations (annotées) du prélude actuel du compilateur Tiger (`prelude.tih`).

```
/* Print STRING on the standard output. */
primitive print (string: string)
/* Note: this is an EPITA extension. Same as print(), but the output
   is written to the standard error. */
primitive print_err (string: string)
/* Note: this is an EPITA extension. Output INT in its decimal
   canonical form (equivalent to "%d" for printf()). */
primitive print_int (int: int)
/* Flush the output buffer. */
primitive flush ()
/* Read a character on input. Return an empty string on an end of
   file. */
primitive getchar () : string

/* Return the ascii code of the first character in STRING and -1 if
   the given string is empty. */
primitive ord (string: string) : int
/* Return the one character long string containing the character which
   code is CODE. If CODE does not belong to the range [0..255], raise
   a runtime error: "chr: character out of range." */
primitive chr (code: int) : string
/* Return the size in characters of the STRING. */
primitive size (string: string) : int

/* Note: this is an EPITA extension. Return 1 if the strings A and B
   are equal, 0 otherwise. Often faster than strcmp() to test string
   equality. */
primitive streq (s1: string, s2: string) : int
/* Note: this is an EPITA extension. Compare the strings A and B:
   return -1 if A < B, 0 if equal, and 1 otherwise. */
primitive strcmp (s1: string, s2: string) : int
/* Return a string composed of the characters of STRING starting at
   the FIRST character (0 being the origin), and composed of LENGTH
   characters (i.e., up to and including the character
   FIRST + LENGTH). */
primitive substring (string: string, start: int, length: int) : string
/* Concatenate FIRST and SECOND. */
primitive concat (first : string, second: string) : string

/* Return 1 if BOOLEAN = 1, else return 0. */
primitive not (boolean : int) : int

/* Exit the program with exit code STATUS. */
primitive exit (status: int)
```