Correction du partiel CMP1

EPITA – Apprentis promotion 2013
Tous documents (notes de cours, polycopiés, livres) autorisés.
Calculatrices et ordinateurs interdits.

Mars 2011 (1h30)

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

Best-of: Le *best of* est tiré des copies des étudiants ; les fautes de français aussi, le cas échéant.

Lisez bien les questions, chaque mot est important. Écrivez court, juste, et bien ; servez vous d'un brouillon. Une argumentation informelle mais convaincante est souvent suffisante. Gérez votre temps, ne restez pas bloqués sur les questions les plus difficiles. Une lecture préalable du sujet est recommandée.

Cette correction contient 15 pages. Les pages 1–9 contiennent l'épreuve et son corrigé. Ce document comporte en pages 10–12 une enquête (facultative) sur le cours et le projet Tiger y répondre sur la feuille de QCM ci-jointe). Enfin, les pages 13–15 sont dévolues aux annexes.

1 Bison

Voici une grammaire au format EBNF (Extended BNF).

```
exps ::= [ exp { ',' exp }]
```

1. Qu'exprime cette grammaire ?

   Correction: Une suite arbitrairement longue de `exps` séparées par des '`,`', éventuellement vide. Bien voir que le '`,`' est ici un séparateur (comme en Pascal ou en Tiger), pas un terminateur (comme en C).

2. Que signifie BNF ?

   Correction: Backus-Naur Form. Il s'agit d'une notation permettant de décrire des grammaires hors-contexte.
   Note : D'après Wikipédia, les règles EBNF doivent être terminées par un point-virgule (`http://en.wikipedia.org/wiki/Extended_Backus_Naur_Form`). Je n'ai pas suivi ce style ici pour ne pas introduire de confusion avec la grammaire figurant dans le manuel de référence du compilateur Tiger, qui n'utilise pas non plus de point-virgule terminateur.
   Étonnamment, il y a eu très peu de réponses correctes à cette question, sachant que BNF a déjà été évoqué en cours de THL.

   Best-of:
   – Bison
   – Bison Native Format
   – Basic Notation Form
   – Forme Normale d'Automate
   – BNF est une grammaire qui ne contient pas le mot vide.
   – Banc Non Fumeur

3. On souhaite implémenter un parser pour cette grammaire avec Bison. Or ce dernier ne sait pas traiter des grammaires EBNF. Traduisez la grammaire précédente en une grammaire Bison.

   Correction: Il suffit d'exprimer la répétition { ',' exp } en utilisant deux règles et une récurrence. Prendre soin d'utiliser la récurrence à gauche, qui permet de limiter la taille de la pile utilisée.

```
exps  :
  /* Empty. */
| exps.1
;

exps.1 :
  exp
| exps.1 ',' exp
;
```

4. Complétez votre proposition en incluant les actions sémantiques, permettant de construire un arbre de syntaxe abstraite (AST). Le type du nœud correspondant au symbole `exp` est `exp_t`. Vous êtes libres (de réfléchir) quant au choix du type de nœud pour `exps`.

   Correction: On décide d'utiliser une liste de la bibliothèque standard du C++ pour représenter une `exps`.

```
exps  :
  /* Empty. */ { $$ = new std::list<exp_t*>; }
| exps.1        { $$ = $1; }
;

exps.1 :
  exp           { $$ = new std::list<exp_t*> (1, $1); }
| exps.1 ',' exp { $$ = $1; $$->push_back ($3); }
;
```

5. Quelle(s) règle(s) faut-il ajouter à votre grammaire Bison pour faire de la reprise sur erreur ?

   Correction: Une règle comme celle-ci, qui va consommer l'entrée jusqu'à ce qu'une virgule soit trouvée, puis tenter de reprendre l'analyse après celle-ci.

```
exps.1 :
  error ',' exp  { $$ = new std::list<exp_t*> (1, $3); }
;
```

2 Tiger

Considérez le bout de code Tiger ci-dessous.

```
type tree = { left : tree, val : int, right : tree }

function node (l : tree, v : int, r : tree) : tree =
 tree { left = l, val = v, right = r }

function leaf (v : int) : tree =
 tree { left = nil, val = v, right = nil }

var t := node (node (leaf (1),
                     2,
                     leaf(3)),
               4,
               node (leaf (5),
                     6,
                     leaf (7)))
```

1. À quoi peut bien servir `tree` ?

   Correction: À implémenter un arbre binaire (d'entiers), par exemple pour stocker un ensemble de valeurs, un peu comme l'implémentation de `std::set` en C++.

2. Quel nom pourrait-on donner à `node` et `leaf` ?

   Correction: (La question était un peu ambiguë : une meilleure formulation aurait été : « Comment pourrait-on qualifier le rôle `node` et `leaf` ? ».)
   La réponse était attendue était : le rôle de *constructeurs* (externes), puisque ces routines servent à construire des `node` et `leaf` respectivement.

3. Citer les différentes phases de la partie frontale (*front end*) d'un compilateur comme `tc`.

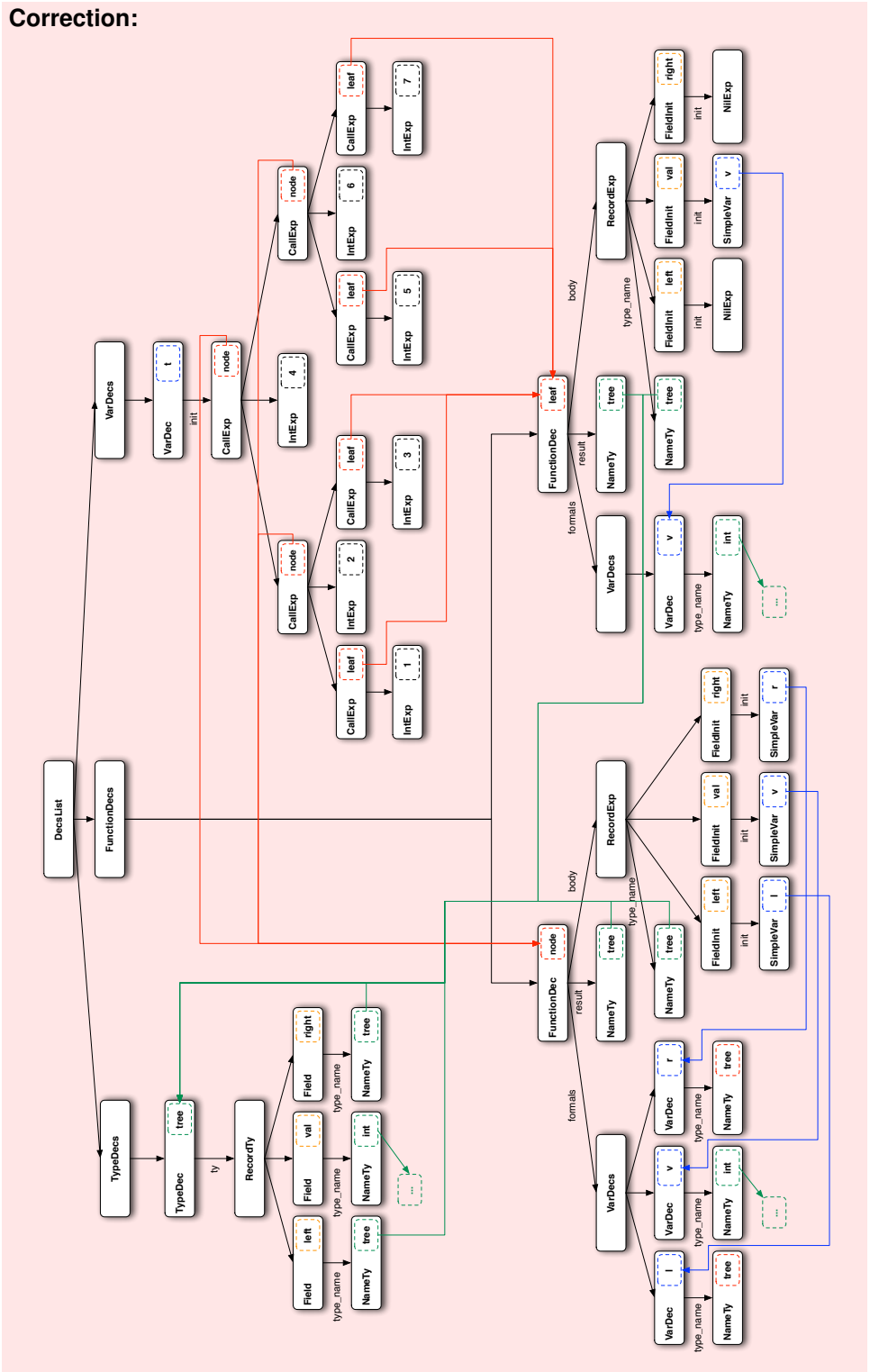
   Correction: (i) scanner, (ii) parser et construction de l'AST, (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

4. Quelle différence faites-vous entre un arbre de dérivation et un arbre de syntaxe abstraite (ou Abstract Syntax Tree (AST)) ?

   Correction: L'arbre de dérivation ou *parse tree* est un produit direct du parser, et reproduit fidèlement la syntaxe concrète du programme en entrée. Un AST est une représentation synthétique du programme, débarrassée des artefacts de la syntaxe concrète.
   On trouvera donc dans un arbre de dérivation des éléments qui sont absents de la syntaxe abstraite :
   - tous les non terminaux intermédiaires faisant apparaître toutes les productions utilisées lors de l'analyse syntaxique (par ex. : `program` → `exp` → `INT`) ;
   - la ponctuation : virgules, points-virgules (séparateurs ou terminateurs), deux-points, parenthèses, crochets, etc.
   - le sucre syntaxique, éventuellement (partiellement) supprimé dans l'AST : opérateurs de logique booléens, `if-then` sans `else`, moins unaire, etc.
   - le sel syntaxique, c'est-à-dire des éléments de syntaxe qui pourraient ne pas faire partie de l'expression d'un programme, et qui permettent (essentiellement) à l'utilisateur de mieux repérer ses erreurs et aux outils de l'aider dans cette tâche. Les mots-clefs terminateurs du Bourne Shell (`fi`, `done`, `esac`, etc.) remplissent cette fonction (en sus de simplifier l'analyse du langage).

5. Donnez une représentation de l'AST du programme Tiger ci-dessus. Aidez-vous de l'annexe A à la fin de ce document.

Correction:



6. Sur le schéma de la question précédente, indiquez les liens entre sites d'utilisation et sites de définition à l'aide de flèches partant des premiers et allant vers les seconds.

Correction: Voir la réponse précédente. On notera que les nœuds 'NameTy int' n'ont pas de cible réellement matérialisée, puisque c'est la sémantique du langage qui fournit cette liaison « par défaut ».

7. Le type `tree` est défini récursivement ; décrivez le mécanisme de la partie chargée du calcul des liaisons dans le compilateur qui autorise une telle définition récursive.

Correction: Lors de l'analyse des noms, chaque bloc de déclaration homogène (c'est-à-dire, composé uniquement de déclarations successives de types ou uniquement de déclarations successives de fonctions) est analysé en deux passes :
(a) les en-têtes (noms) des déclarations sont d'abord visités, et insérés dans l'environnement correspondant (table des symboles de types ou de fonctions) ;
(b) puis les corps (définitions) des déclarations sont parcourus.
Ainsi, une définition d'un type (comme celle de `tree`) peut faire référence au nom de ce type, ou à celui d'un autre type défini dans le même bloc homogène.

8. Écrivez une fonction Tiger `print_tree` prenant un `tree` en argument et affichant ses éléments comme dans la sortie ci-dessous (avec `t` passé en argument) :

```
␣␣␣␣1
␣␣2
␣␣␣␣3
4
␣␣␣␣5
␣␣6
␣␣␣␣7
```

Le niveau d'indentation (marqué par des paires d'espace) représentant la « profondeur » de chaque valeur d'entier par rapport à la racine (l'élément passé en argument à `print_tree`). L'annexe B peut vous être utile.

Correction: Une solution possible, utilisant la récursivité :

```
function print_tree (t : tree) =
 let
   function print_rec (t : tree, indent : string) =
     if (t <> nil) then
       (
         print_rec (t.left,  concat (indent, "  "));
         print (indent); print_int (t.val); print ("\n");
         print_rec (t.right, concat (indent, "  "))
       )
 in
   print_rec (t, "")
 end
```

Attention, beaucoup de réponses ont omis le cas où l'argument de `print_tree` est une valeur `nil`.

9. Écrire une fonction `incr` prenant un `tree` en argument appliquant cette fonction à tous les `int` contenus dans le `tree`, et renvoyant un `tree` structurellement équivalent à celui d'entrée, mais dont tous les éléments entiers ont été incrémentés de 1. Par exemple, 'print_tree (incr (t))' afficherait

```
␣␣␣␣2
␣␣3
␣␣␣␣4
5
␣␣␣␣6
␣␣7
␣␣␣␣8
```

Correction:

```
function incr (t : tree) : tree =
 if t = nil then
   nil
 else
   node (incr (t.left), t.val + 1, incr (t.right))
```

Là aussi, le cas où `t` vaut `nil` a souvent été oublié.

10. On aimerait pouvoir généraliser la fonction `incr` de la question précédente et écrire une fonction `map` prenant un `tree` en argument et une fonction de type `int` → `int` (prenant un entier, renvoyant un entier), appliquant cette fonction à tous les `int` contenus dans le `tree`, et renvoyant un `tree` structurellement équivalent à celui d'entrée, mais portant les résultats de la fonction appliquée.

    (a) Pourquoi n'est-ce pas possible en l'état avec notre langage Tiger ?

    Correction: Parce que Tiger ne comporte pas de types fonctions, et ne sait pas manipuler une fonction comme une valeur quelconque.

    Best-of: En raison de la gestion des portées.

    (b) En modifiant légèrement l'énoncé du problème, il est possible de s'en sortir grâce à la programmation orientée objet (toujours en Tiger). Comment ?

    Correction: En utilisant des objets-fonctions (ou *functors*), c'est-à-dire des objets se comportant comme des fonctions. En C++, c'est d'autant plus simple à utiliser que l'on peut surcharger l'opérateur « parenthèses » (`operator()`) pour utiliser un objet-fonction comme une fonction. Tiger ne fournit pas un tel sucre syntaxique, mais on peut toujours bénéficier de la fonctionnalité en utilisant un nom de méthode classique (comme `eval`, `go`, `apply`, `run`, etc.).
    La fonction `map` prendra donc un objet-fonction `f` en second argument, en lieu et place de la fonction initialement souhaitée. Pour garantir la réutilisabilité de `map`, on prendra soin de donner à `f` un type abstrait, de sorte que l'on puisse passer des instances de types différents en argument. Voir la correction de la question suivante pour plus de détails.

    Best-of: En utilisant le binding.

    (c) En utilisant la réponse à la question 10b, écrivez la fonction `map`, y compris ce qu'il faut ajouter pour que l'ensemble fonctionne.

Correction: Commençons par définir une classe « abstraite » définissant l'interface d'un functor, dont le code est contenu dans une méthode `eval` Tiger ne permet pas de définir de méthode abstraite (« virtuelle pure » en C++), aussi devons-nous donner un code fictif pour respecter le typage. On pourrait envisager d'insérer un appel à `exit` (avec un argument différent de 0) pour déclencher une erreur au *run time*, et ainsi « interdire » l'usage de cette méthode abstraite.

```
class Functor
{
 /* Dummy (identity). */
 method eval (i : int) : int = i
}
```

Ensuite, on peut définir des sous-classes de `Foreign` définissant des objets-fonctions concrets, par exemple incrémentant leur argument ou les élevant au carré. On rappelle que toutes les méthodes sont virtuelles (polymorphes) en Tiger, donc aucun mot-clef (`virtual`, `deferred`, etc.) n'est nécessaire pour bénéficier du polymorphisme.

```
class Incr extends Functor
{
 method eval (i : int) : int = i + 1
}

class Square extends Functor
{
 method eval (i : int) : int = i * i
}
```

Enfin, nous pouvons écrire `map` de façon récursive, en utilisant la méthode `eval` du functor `f` passé en argument.

```
function map (t : tree, f : Functor) : tree =
 if t = nil then
  nil
 else
  node (map (t.left, f),
        f.eval (t.val),
        map (t.right, f))
```

11. **Bonus** Le type du champ `val` d'un `tree` est fixe : il s'agit toujours d'un `int`. C'est dommage, car peu réutilisable ; imaginons que l'on ait besoin d'une structure similaire à `tree`, mais avec des `string` (ou tout autre type) à la place. Il y a au moins deux solutions à ce problème. La première solution, que l'on nommera $\mathcal{A}$, est déjà implémentable dans le langage Tiger utilisé à l'EPITA [1], mais présente des défauts. La seconde solution, meilleure du point de vue du typage et des performances, et que l'on nommera $\mathcal{B}$, serait classique à implémenter en C++, mais n'est pas réalisable en l'état en Tiger.

    (a) Quelles sont ces deux solutions ?

1. On rappelle que le langage Tiger utilisé à l'EPITA est différent de celui que propose Andrew Appel, et comporte certaines additions.

Correction: L'objectif de cette question est de rendre `tree` *générique* pour en faire un conteneur capable de stocker autre chose que des `int`s.
La solution $\mathcal{A}$ s'appuie sur la programmation orientée objet. Elle permet d'introduire dans `tree` une généricité *dynamique* (qui s'exprimer à l'exécution). Au lieu de stocker des `int`s, tree va stocker des instances d'une classe abstraite, comme `Object` par exemple[a].
La solution $\mathcal{B}$ s'appuie sur la programmation générique et les types paramétrés (templates de classes en C++). Cette solution apporte à `tree` une généricité *statique* (résolue à la compilation). Cette fonctionnalité n'existe pas dans le langage Tiger utilisé à l'EPITA. Il est cependant facile d'imaginer une extension au langage qui permette de lui ajouter des types paramétrés.

a. Bien qu'`Object` ne soit pas un choix très pertinent, puisque Tiger ne dispose pas d'un opérateur de conversion dynamique vers le type exact d'un objet (comme `dynamic_cast` en C++). Une classe capable d'être « visitée » par un visiteur serait un meilleur choix.

(b) Les deux solutions $\mathcal{A}$ et $\mathcal{B}$ de la question précédente s'appuient chacune sur un type de polymorphisme. Quel est celui correspondant à $\mathcal{A}$? et celui relatif à $\mathcal{B}$?

Correction: Le polymorphisme relatif à la solution $\mathcal{A}$ (OO) est le *polymorphisme d'inclusion* : `tree` peut contenir des instances de sous-classes du type utilisé pour le champ `val` de `tree`.
Le polymorphisme correspondant à la solution $\mathcal{B}$ (programmation générique) est le *polymorphisme paramétrique* : `tree` peut contenir des valeurs du type de son paramètre.

(c) Réécrivez les définitions de `tree`, `node` et `leaf` en utilisant la solution $\mathcal{A}$.

Correction: On introduit d'abord une classe abstraite `Element` ainsi que le `Visitor` associé

```
function abort () = exit (1)

class Element
{
  method print () = ()
  method clone () : Element = new Element
  method accept (v : Visitor) = abort ()
}

class Visitor
{
  method visit_Integer (e : Integer) = ()
}
```

Correction: On peut ensuite sous-classer `Element` pour créer un type de valeur à stocker dans l'arbre, comme des entiers, grâce à la classe `Integer` :

```
class Integer extends Element
{
  var val := 0
  method clone () : Element =
    let var res := new Integer
    in (res.val = self.val; res) end
  method print () = print_int (self.val)
  method accept (v : Visitor) = v.visit_Integer (self)
}

/* Integer construction helper. */
function integer (i : int) =
  let var res := new Integer in (res.val = i; res) end
```

On change ensuite le type `tree` pour qu'il stocke des `Elements` au lieu d'`ints`, ainsi que les « constructeurs » `node` et `leaf` :

```
type tree = { left : tree, val : Element, right : tree }

function node (l : tree, v : Element, r : tree) : tree =
  tree { left = l, val = v, right = r }

function leaf (v : Element) : tree =
  tree { left = nil, val = v, right = nil }
```

Une opération sur un élément stocké dans un arbre nécessite un visiteur concret. Par exemple, la classe `Incrementer` incrémente un `Integer` :

```
class Incrementer extends Visitor
{
  method visit_Integer (e : Integer) =
    e.val := e.val + 1
}
```

Concernant l'application de fonctions à l'aide de `map` (cf. question 10c), il est inutile de changer le code de celle-ci. En revanche, les foncteurs doivent être modifiés pour travailler sur des `Elements` :

```
class Functor
{
  /* Dummy. */
  method eval (e : Element) : Element = (abort (); e)
}

class Incr extends Functor
{
  method eval (e : Element) : Element =
    let var res := e.clone () in
      (e.accept(new Incrementer); e)
    end
}
```

(Idem pour une routine comme `print_tree`.)

3 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire. Les renseignements ci-dessous ne seront bien entendu pas utilisés pour noter votre copie. Ils ne sont pas anonymes, car nous souhaitons pouvoir confronter réponses et notes. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Sauf indication contraire, vous pouvez cocher plusieurs réponses par question. Répondez sur la feuille de QCM qui vous est remise. N'y passez pas plus de dix minutes.

Cette épreuve

1. Sans compter le temps mis pour remplir ce questionnaire, combien de temps ce partiel vous a-t-il demandé (si vous avez terminé dans les temps), ou combien vous aurait-il demandé (si vous aviez eu un peu plus de temps pour terminer) ?
   A Moins de 30 minutes.
   B Entre 30 et 60 minutes.
   C Entre 60 et 90 minutes.
   D Entre 90 et 120 minutes (estimation).
   E Plus de 120 minutes (estimation).
2. Ce partiel vous a paru
   A Trop difficile.
   B Assez difficile.
   C D'une difficulté normale.
   D Assez facile.
   E Trop facile.

Le cours

3. Vous avez pris des notes
   A Aucune.
   B Sur papier.
   C Sur ordinateur à clavier.
   D Sur ardoise numérique.
   E Sur le journal du jour.
4. Quelle a été votre implication dans les cours CMP1 ?
   A Rien.
   B Bachotage récent.
   C Relu les notes entre chaque cours.
   D Fait les annales.
   E Lu d'autres sources.
5. 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.

6. 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.
7. La charge générale du cours en sus de la présence en amphi (relecture de notes, compréhension, recherches supplémentaires, etc.) est
   A Telle que je n'ai pas pu suivre du tout.
   B Lourde (plusieurs heures par semaine).
   C Supportable (environ une heure de travail par semaine).
   D Légère (quelques minutes par semaine).

Les formateurs

8. 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.
9. Les assistants
   A Ne sont pas pédagogues.
   B Parlent à des étudiants qui sont au dessus de mon niveau.
   C M'ont aidé à avancer dans le projet.
   D Ont résolu certains de mes gros problèmes, mais ne m'ont pas expliqué comment ils avaient fait.
   E Pourraient viser plus haut et enseigner des notions supplémentaires.

Le projet Tiger

10. Vous avez contribué au développement du compilateur de votre groupe (une seule réponse attendue) :
    A Presque jamais.
    B Moins que les autres.
    C Équitablement avec vos pairs.
    D Plus que les autres.
    E Pratiquement seul.
11. La charge générale du projet Tiger est
    A Telle que je n'ai pas pu suivre du tout.
    B Lourde (plusieurs jours de travail par semaine).
    C Supportable (plusieurs heures de travail par semaine).
    D Légère (une ou deux heures par semaine).
    E J'ai été dispensé du projet.

12. Y a-t-il de la triche dans le projet Tiger ? (Une seule réponse attendue.)
    A Pas à votre connaissance.
    B Vous connaissez un ou deux groupes concernés.
    C Quelques groupes.
    D Dans la plupart des groupes.
    E Dans tous les groupes.

Questions 13-19 Le projet Tiger vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille qui suit. (Une seule réponse attendue par question.)

A Pas du tout.
B Trop peu.
C Correctement.
D Bien.
E Très bien.

13. Formation au C++.
14. Formation à la modélisation orientée objet et aux *design patterns*.
15. Formation à l'anglais technique.
16. Formation à la compréhension du fonctionnement des ordinateurs.
17. Formation à la compréhension du fonctionnement des langages de programmation.
18. Formation au travail collaboratif.
19. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 20-24 Comment furent les étapes du projet ? Répondre selon la grille suivante. (Une seule réponse attendue par question ; ne pas répondre pour les étapes que vous n'avez pas faites.)

A Trop facile.
B Facile.
C Nickel.
D Difficile.
E Trop difficile.

20. Rush `.tig` : mini-projet en Tiger (Bistromatig).
21. `TC-0`, Scanner & Parser.
22. `TC-1`, Scanner & Parser, Tâches, Autotools.
23. `TC-2`, Construction de l'AST et pretty-printer.
24. `TC-3`, Liaison des noms et renommage.

A Classes de la syntaxe abstraite du langage Tiger

Voici une copie du fichier 'src/ast/README' fourni avec le code de tc.

```
Tiger Abstract Syntax Tree nodes with their principal members.
Incomplete classes are tagged with a '*'.

/Ast/               (Location location)
  /Dec/             (symbol name)
    FunctionDec     (VarDecs formals, NameTy result, Exp body)
      MethodDec     ()
    RuleDec         (Exp match, Exp build)
    TypeDec         (Ty ty)
    VarDec          (NameTy type_name, Exp init)

  /Exp/             ()
*   /Var/
      CastVar       (Var var, Ty ty)
*     FieldVar
      SimpleVar     (symbol name)
      SubscriptVar  (Var var, Exp index)

*   ArrayExp
*   AssignExp
*   BreakExp
*   CallExp
*     MethodCallExp
    CastExp         (Exp exp, Ty ty)
    ForExp          (VarDec vardec, Exp hi, Exp body)
*   IfExp
    IntExp          (int value)
*   LetExp
    MetavarExp      ()
    NilExp          ()
*   ObjectExp
    OpExp           (Exp left, Oper oper, Exp right)
*   RecordExp
*   SeqExp
*   StringExp
    WhileExp        (Exp test, Exp body)

  /Ty/              ()
    ArrayTy         (NameTy base_type)
    ClassTy         (NameTy super, DecsList decs)
    NameTy          (symbol name)
*   RecordTy

  DecsList          (decs_type decs)

  Field             (symbol name, NameTy type_name)

  FieldInit         (symbol name, Exp init)
```

```
Some of these classes also inherit from other classes.

/Escapable/
  VarDec            (NameTy type_name, Exp init)

/Metavariable/      (unsigned id)
  MetavarExp        ()

/Typable/
  /Dec/             (symbol name)
  /Exp/             ()
  /Ty/              ()

/TypeConstructor/
  /Ty/              ()
  FunctionDec       (VarDecs formals, NameTy result, Exp body)
  TypeDec           (Ty ty)
```

B 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 (a: string, b: 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 (a: string, b: 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, first: 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)
```