Correction du partiel CMP1

EPITA – Promo 2012
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices et ordinateurs interdits.

Janvier 2010 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

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. Une lecture préalable du sujet est recommandée.

1 Warm Up

1. Comment faire comprendre à votre scanner Flex que l'entrée Tiger '`for`' doit produire un token FOR et non un token ID (correspondant au symbole '`for`') ?

   Correction: Les règles Flex générant respectivement le FOR et le token ID reconnaissent toutes les deux l'entrée '`for`'. Pour que la première prenne le pas sur la seconde, il suffit de la placer avant cette dernière dans le fichier d'entrée Flex : par définition, le scanner la sélectionnera.
   Se limiter à dire qu'il est nécessaire de définir un motif pour explicitement attraper la chaîne `for` était insuffisant.

   Best-of:
   – Une solution est de déclarer au préalable tous les tokens ID et leur donner un nom. De cette façon le token FOR et les tokens ID seront différenciés.
   – `%for FOR`
   – `left id`
   `left for`
   – Il suffit de regarder quels tokens suivent le token FOR. En fonction de ces tokens on peut déterminer que l'entrée Tiger '`for`' doit produire un token FOR et non un ID.
   – En surchargeant le symbole '`for`'.

2. Pourquoi est-ce que votre scanner Flex génère à partir de l'entrée Tiger '`<=`' un seul token LE (*lower or equal*) et non un token LT (*lower than*) suivi d'un token EQ (*equal*) ?

   Correction: Parce qu'un scanner Flex cherche à attraper le motif le plus long (*longest match*). (En cas d'égalité entre deux règles, c'est le principe de la question précédente qui s'applique.)
   Attention, sans cette explication, il est incorrect de répondre uniquement que la présence ou l'absence d'espace entre les deux tokens suffit pour désambiguïser. Certains m'ont parlé d'associativité ; cela n'a pas de sens, la question traitait de de problèmes de *scanner* et non de *parser*.

Best-of:
- Il ne produit qu'un sel token car les règles LT et EQ sont utilisées simultanément.
- Dans la construction de l'AST, il est nécessaire d'avoir un seul token LE et non deux tokens LT et EQ.
- Étant donné que le scanner Flex regarde d'un symbole vers l'avant, il sait que '<=' ne doit créer qu'un seul token et non deux. En effet, le scanner repose sur l'algorithme LALR(1).
- Flex génère un '<=' car le '<' est prioritaire sur le '='.
- Le sucre syntaxique est la liaison des noms.
- Plus simple à gérer dans la grammaire.

3. 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:
- [. . .] sucre synthaxique [. . .]
- [. . .] synthaxe [. . .]
- C'est du syntaxique pur.
- Le sucre syntaxique est la dernière étape du compilateur pour passer en langage assembleur.
- Le sucre synthaxique est l'ensemble des conditions permettant de distinguer les priorités entre les non-terminaux (`%nonassoc` et `%left`) [. . .]
- Le sucre syntaxique ce sont des petits mots qui sont ajoutés à la grammaire.
- Un moyen compliqué de faire quelquechose de simple.

4. Pourquoi les références du C++ peuvent-elles être considérées comme du sucre syntaxique ?

Correction: Les références sont essentiellement des raccourcis d'écritures pour les pointeurs. Les bénéfices de ce sucre syntaxique sont la concision (par d'opérateur de déréférencement) et la sûreté (initialisation obligatoire, pas d'arithmétique des pointeurs).

Best-of:
- Les références du C++ peuvent être considérée comme du sucre syntaxique car tout ce que fait le C++, le C pouvait le faire, le C++ permet juste de faire les choses plus facilement.
- Car c'est beau et rapide.
- Le code C++ est un langage qui est traduit par le compilateur en langage machine. D'un certain point de vue c'est du sucre syntaxique puisque l'on pourrait directement ecrire dans ce langage machine.

2 Dessine-moi un programme

Considérez la grammaire suivante, qui s'inspire d'un sous-ensemble du langage Logo.

```
⟨instrs⟩ ::= ⟨instr⟩
           | ⟨instrs⟩ ⟨instr⟩

⟨instr⟩  ::= forward num
           | left num
           | right num

           # Loop.
           | repeat num [ ⟨instrs⟩ ]

           # Subprogram definition.
           | to id ⟨instrs⟩ end

           # Subprogram call.
           | id
```

Dans cette grammaire, le terminal num désigne un entier littéral et le terminal id un identifiant (qui suit les mêmes règles lexicales que les identifiants du langage Tiger).

Le langage Logo permet (entre autres) de dessiner sur un écran à l'aide d'une « tortue » que l'on manipule avec des instructions comme `forward 10` (avance de 10 unité en traçant un trait) ou `right 30` (tourne à droite de 30 degrés).

1. Est-ce que cette grammaire est LL(k) et si oui, pour quel k ? Pourquoi ?

   Correction: La production instrs comporte une récurrence à gauche, donc cette grammaire n'est pas LL(k), pour n'importe quel k.

   Best-of:
   – Cette grammaire est LL(0).
   – Cette grammaire est LL(k) pour $k = 1$ car les instructions sont linéaires à droite.
   – C'est bien un LL(k) avec $k = 2$ [. . .]
   – Elle est LL(3) [. . .]
   – Non, elle n'est pas LL(k) car elle n'est pas exclusivement récursive à gauche.
   – [. . .] De plus elle est clairement récursive à droite et non ambigüe.
   – Le langage n'est pas associatif à gauche donc il n'est pas LL.
   – Les FIRST de `<instr>` sont { "r", "i", "t", "l", "f" }.

2. Est-elle LR(k) et si oui, pour quel k ? Justifiez votre réponse.

   Correction: Cette grammaire est LR(0) : elle ne comporte aucune ambiguïté et on peut facilement en déduire un automate LR(0) (laissé en exercice au lecteur).

Best-of:
- Elle est à la limite LR 2 [. . .]
- Non, puisqu'elle est LL(1).
- Elle n'est pas LR, car la dérivation n'est pas à droite.
- Elle ne peut être LR(k) car elle est LL(1).
- [. . .] Un look-above d'un token en arrière permet de déterminer la règle courante et laquelle suivre.
- Elle n'est pas LR car elle ne peut pas être lue de droite à gauche.
- Cette grammaire n'est pas LR car on ne peut pas remonter dans les règles,donc je pencherai pour un $k = -42$.
- Non, car si on execute de droite à gauche, on arrivera pas à la même chose que de gauche à droite. Exemple : avancer, droit $\neq$ droit puis avance.
- À l'inverse, on a une LR(k) pour $k \geq 2$ [. . .]
- Cette grammaire est LR(k) pour $k = 4$.

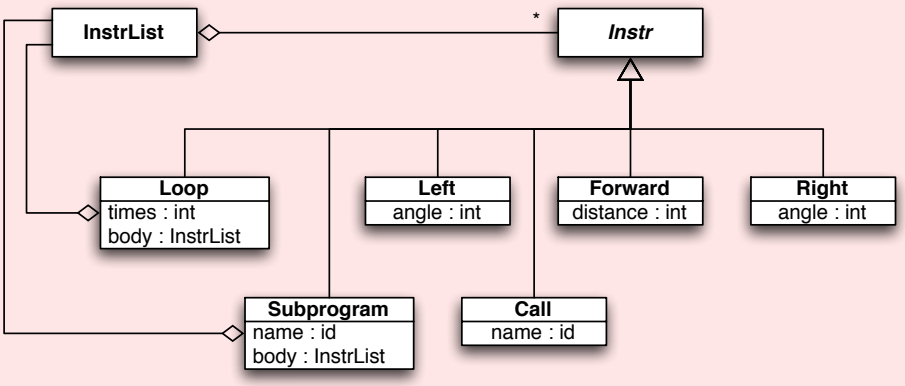
3. On décide d'implémenter un parser Bison pour ce langage. Quel(s) changement(s) éventuel(s) pensez-vous apporter à cette grammaire lors de cette opération ?

Correction: Aux détails de syntaxe Bison près, aucun. Cette grammaire est LR(1) et écrite en BNF : la traduire en code Bison est immédiat.
Beaucoup de copies proposent de transformer la récurrence à gauche de la production instrs en une récurrence à droite ; c'est à la fois inutile et contre-performant, puisque l'on sait que les analyseurs LR préfèrent les récurrences à gauche, qui réduisent au plus tôt, et évitent de faire grossir la (les) pile(s) de l'automate.

Best-of:
- Modifier les non-terminaux en token.
- La portée des identifiants risque de créer des ambiguïtés.

4. Les arbres de syntaxe abstraite (ASTs) produits par Bison seront implémentés sous forme d'objets C++, à l'instar de l'implémentation des ASTs de `tc`. Donnez une description synthétique des classes représentant les différents nœuds de la syntaxe abstraite du langage étudié dans cet exercice, soit sous forme de diagramme de classes UML, soit sous forme de code C++.

Correction: Voici une possibilité :



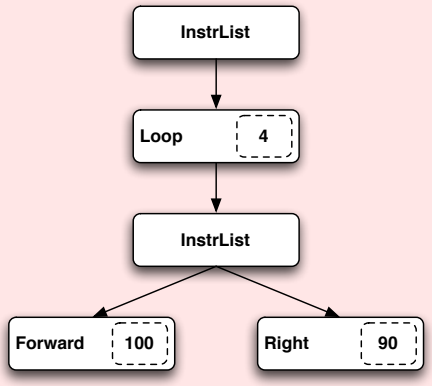
Notez que l'on peut factoriser les nœuds `Forward`, `Left` et `Right` en une unique classe `Move`, comportant une étiquette (implémentée avec une `enum` par exemple) permettant d'identifier la nature du mouvement.

5. Voici un programme Logo qui trace un carré de 100 unités de côté :

```
repeat 4 [ forward 100 right 90 ]
```

Donnez une représentation arborescente (graphique ou textuelle) de la syntaxe abstraite de ce programme.

Correction: En reprenant les noms des nœuds de la réponse à la question 4 :



ou encore :

```
InstrList (Loop (4, InstrList (Forward (100), Right (90))))
```

Attention, pour cette question (et pour la suivante), j'ai trouvé des réponses qui faisaient mention d'éléments typiques de la syntaxe concrète et qui n'auraient pas dû apparaître (comme les crochets) : il était bien question de syntaxe abstraite, on voulait donc voir des AST, pas des arbres de dérivation (*parse trees*).

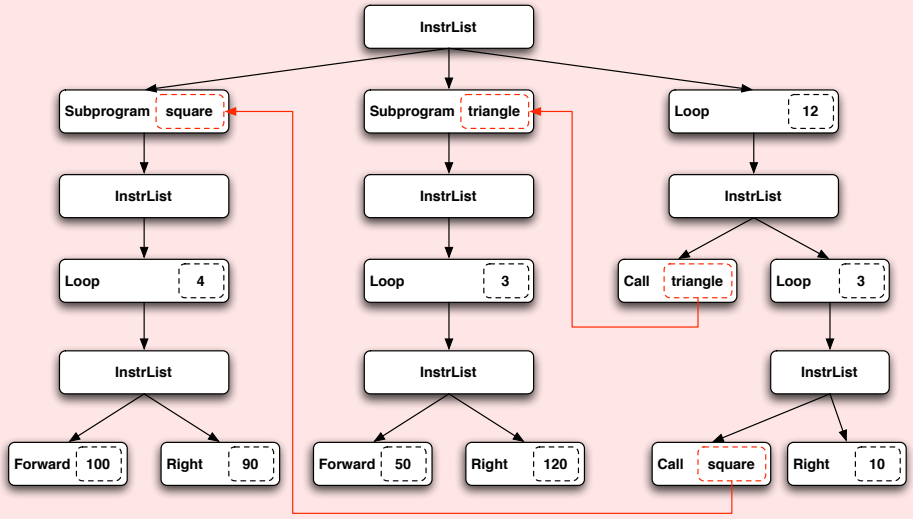
Best-of:

– Position de départ, on trace une ligne de 100 unités, on tourne de 90 degrés, puis on trace une ligne de 100 unités, on tourne de 90 degrés, puis on trace une ligne de 100 unités, on tourne de 90 degrés, puis on trace une ligne de 100 unités.
Enfin, on tourne de 90 degrés !
Fin du programme.

6. Même exercice avec le programme suivant :

```
to square
repeat 4 [ forward 100 right 90 ]
end
to triangle
repeat 3 [ forward 50 right 120 ]
end
repeat 12 [ triangle repeat 3 [ square right 10 ] ]
```

Correction: De même :

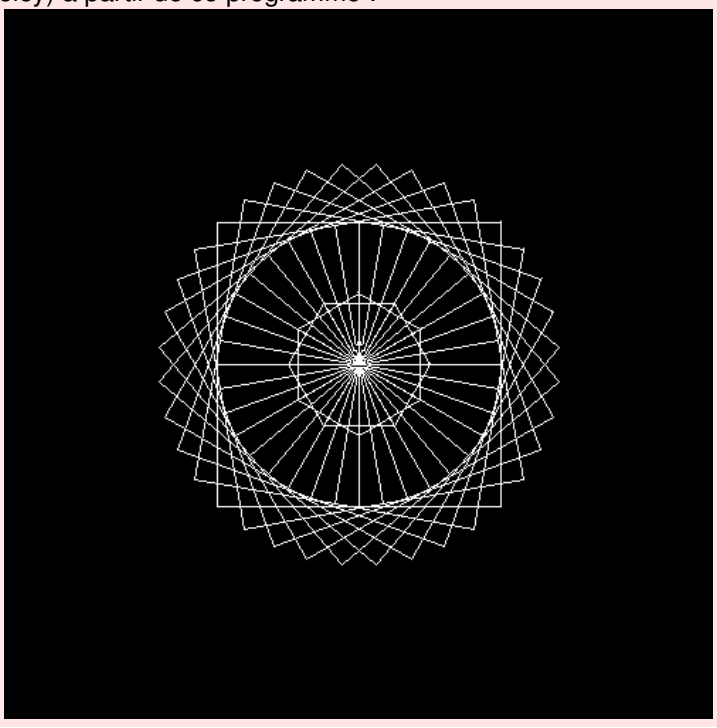


ou encore :

```
InstrList (Subprogram (square,
                       InstrList (Loop (4, InstrList(Forward (100),
                                                     Right (90)))),
           Subprogram (triangle,
                       InstrList (Loop (3, InstrList(Forward (50),
                                                     Right 120)))),
           Loop (12,
                 InstrList (Call (triangle),
                            Loop (3, InstrList (Call square
                                                Right 10)))))
```

Attention, dans beaucoup de copies, les définitions des sous-programmes `square` et `triangle` sont mises en ligne (*inlined*) aux sites d'utilisations : ce n'était pas demandé ! On souhaitait un AST conforme à l'entrée, pas une version transformée.

Pour information, voici ce que dessine `ucblogo` (une implémentation de Logo de l'UC Berkeley) à partir de ce programme :



7. Intéressons-nous à l'analyse des noms du programme de la question précédente. Sur le schéma de votre réponse à la question 6, indiquez par des flèches (de préférence en utilisant une couleur différente) les liens entre les sites d'utilisations de noms de sous-programmes et les sites de définitions correspondant.

   Correction: Les liens figurent en rouge sur le diagramme de la réponse précédente.
   Attention au sens des flèches : la logique veut qu'elles partent des sites d'utilisations vers les sites de définitions.

8. La sémantique de ce langage n'a pas été spécifiée très formellement, et rien n'a été dit quant à la portée des identifiants de sous-programmes. Donnez un exemple d'entrée qui pourrait être interprétée comme une ambiguïté sémantique.

   Correction: Il y a au moins deux classes de problèmes :
   – Quid de la portée ? Jusqu'où s'étend l'influence d'un nom de sous-programme ? Est-ce qu'une identifiant peut être redéfini dans une portée intérieure, et masquer (temporairement) son homonyme ?
   – On n'a rien dit au sujet de la récursivité. Est-elle autorisée dans notre langage ? Si oui, faut-il des pré-déclarations (*forward declarations*) ? Oui bien des règles de blocs (*chunks*) comme dans Tiger ?

9. Essayons d'améliorer la spécification sémantique de notre langage pour pallier les problèmes évoqués à la question précédente. Quelle(s) règle(s) relative(s) aux noms de sous-programmes allez-vous ajouter pour lever les ambiguïtés éventuelles évoquées ci-avant ?

   Correction: Plusieurs choix sont valables pour chaque problème, la décision dépend de la forme que l'on souhaite donner au langage et de la complexité que l'on est prêt à introduire dans sa définition (ses spécifications) et son implémentation (son compilateur ou son interprète).
   – On peut ainsi décider
     – d'interdire globalement la redéfinition d'un sous-programme ; ou bien
     – de prendre en compte la notion de portée introduite par une définition de sous-programme, et
       – d'interdire localement (dans la même portée) la redéfinition d'un sous-programme ; ou bien
       – d'autoriser la redéfinition d'un sous-programme si elle ne fait pas partie d'un bloc de définition contiguës (comme dans Tiger) ; ou bien
       – de toujours autoriser la redéfinition d'un sous-programme, et décider que sa dernière définition (dans la portée courante) masque les précédentes ;
       etc.
   – Concernant la récurrence, on peut choisir de l'autoriser ou de l'interdire ; dans ce dernier cas, l'environnement d'exécution nécessitera bien entendu la gestion d'une pile d'appels de sous-programmes.

10. À votre avis, est-il nécessaire d'effectuer une analyse des types après la passe d'analyse des noms ? Justifiez votre réponse.

    Correction: Non : tout est résolu avec le parsing et l'analyse des noms (qui peut elle-même être effectuée lors du parsing).

    Best-of:
    – Il convient toujours de faire une analyse supplémentaire, pour plus de sécurité et pour éviter une erreur syntaxique.
    – Vaut mieux toujours vérifier trop que pas assez !

11. Si l'on souhaite écrire un interprète pour ce langage (travaillant directement à partir des ASTs), quel(s) outil(s) (au sens large) sera (seront) nécessaire(s) à son implémentation ?

Correction: En sus des passes évoquées dans les question précédentes (analyses lexicale et syntaxique, résolution des noms), le seul outil dont nous ayons besoin est celui nous permettant de parcourir les nœuds de l'AST pour en exécuter les instructions. Une telle implémentation est assez directe ; le seul point qui aurait pu présenter des difficultés est l'exécution des sous-programmes, mais l'analyse des noms (qui étiquette les usages par les définitions) a fait l'essentiel du travail.
Il est aussi possible d'« aplatir » l'interprète en effectuant les étapes d'analyses lexicale et syntaxique, de résolution des noms et d'évaluation dans le parser lui-même (interprète « mince »).
Notez que rien n'a été dit quant au support de la récursivité des sous-programmes, mais comme le langage est très pauvre en structures de contrôle, celle-ci serait sans doute peu utile de toute façon. Si l'on décidait quand même de la supporter, il faudrait donc gérer une pile. On peut s'arranger pour que cette pile soit complètement implicite en prenant soin d'implémenter le visiteur-évaluateur de façon récursive : les différentes *activations* de sous-programmes seront autant d'appels de méthodes de visite, gérés par la pile d'appels du langage d'implémentation (le C++ en l'occurrence).

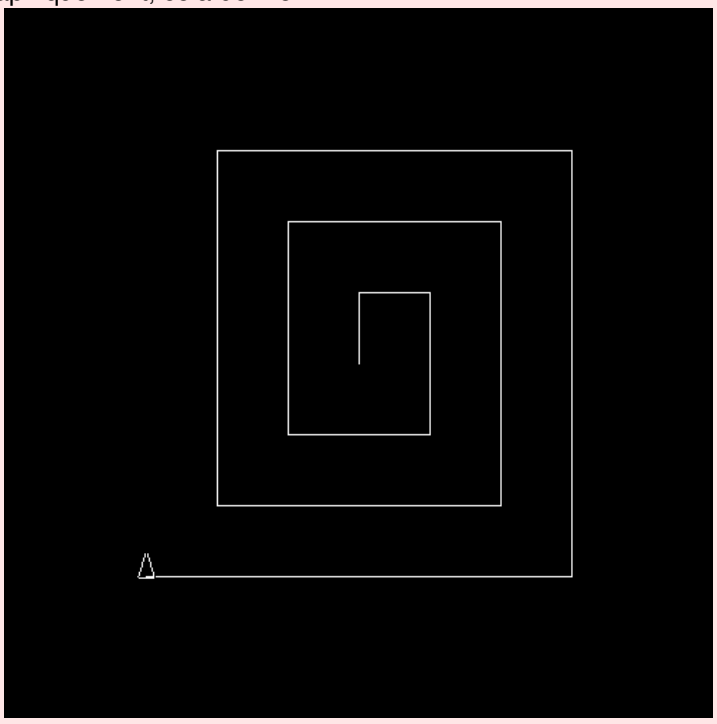
Best-of:
– Boost sera nécessaire à son implémentation.
– Au sens très large : un ordinateur et un clavier.
– Il faudra de bons design patterns.

12. **Bonus.** Écrivez un programme Logo qui dessine une spirale carrée.

Correction: Quelque chose comme :

```
repeat 2 [ forward 50 right 90 ]
repeat 2 [ forward 100 right 90 ]
repeat 2 [ forward 150 right 90 ]
repeat 2 [ forward 200 right 90 ]
repeat 2 [ forward 250 right 90 ]
repeat 2 [ forward 300 right 90 ]
repeat 2 [ forward 350 right 90 ]
```

On sent que le langage proposé est fruste, et qu'il lui manque la notion de variable. Graphiquement, cela donne :



Best-of:

– La syntaxe du langage Logo donnée dans le sujet ne permet de travailler que dans deux dimentions, la création d'une spirale carrée me semble donc impossible.

3 Un peu de C++ pour la route

1. Pourquoi une méthode de classe (fonction membre `static` en C++) ne peut pas être abstraite (`virtual`) ?

   Correction: Une petite erreur dans le sujet : au lieu de lire « `virtual` » il fallait lire « `virtual` pure », qui est l'équivalent de (polymorphe) « abstraite ». « `virtual` » seul signifie seulement polymorphe, mais malgré tout, la question conservait son sens.
   Une méthode polymorphe est une *méthode d'instance* : elle est liée à un objet (entité dynamique), puisque sa résolution (et donc son exécution) dépend(ent) du type exact (dynamique) de ladite instance. Ceci est incompatible avec la notion de *méthode de classe*, qui est attaché à une *classe* (entité statique), et non à un objet. Une telle méthode ne nécessite aucune instance de la classe pour être invoquée (appel de la forme `Class::method()`).
   Il ne fallait pas non plus confondre méthode `static` (méthode de classe, partagée par toutes ses instances) et *fonction* `static` (fonction dont la liaison est interne, c'est-à-dire dont le symbole n'est pas exporté de l'unité de compilation).
   Rappelons également que le code d'une méthode (qu'elle soit `static`, `virtual`, etc.) tourne à l'exécution (*run time*). En effet, en C++ (merci à l'héritage du C pour ce terme limpide ?), `static` ne signifie pas que le code est évalué statiquement – à la compilation (*compile time*)[a].
   Enfin, il faut savoir
   – qu'une méthode de classe est « héritée » : si une classe `A` définit une méthode de classe `m`, et que `B` dérive de `A`, alors `B::m()` est un appel licite (qui exécute le code de `A::m`) ;
   – qu'il est possible de « redéfinir » une méthode de classe (ayant la même signature) dans une sous-classe. Bien sûr, il ne s'agit pas d'une redéfinition au sens de `virtual` : les appels de ces méthodes sont toujours résolus statiquement (à l'instar des appels de méthodes d'instances non virtuelles).

   [a] Contrairement au langage D, où `static` sert précisément à écrire des métaprogrammes évalués statiquement.

   Best-of:
   – Une méthode classe static ne peut pas être abstraite, car lorsqu'elle est appellée, elle sauvegarde ses attributs.
   – Une méthode statique s'éxecute à la compilation, alors qu'une méthode abstraite l'est à l'évaluation.
   – C'est trop abstrait, je n'arrive pas à me projetter.

2. À votre avis, pourquoi une méthode C++ ne peut pas être à la fois `virtual` et `template` ?

Correction: D'une part, le code réel d'une méthode polymorphe (`virtual`) qui est appelée n'est connu qu'à l'exécution, puisqu'il est nécessaire de connaître le type exact (dynamique) de la cible. D'autre part, en C++, une méthode paramétrée (`template`) est *instanciée* (générée) dans une unité de compilation (un fichier objet) pour chaque cas d'utilisation (chaque combinaison unique de paramètres). Ce mécanisme statique dépend donc des *appels* de ladite méthode[a]. Il y a donc là une incompatibilité : une méthode paramétrée est instanciée (à la *compilation*) lorsqu'elle est appelée, or l'appel d'une méthode polymorphe n'est résolu qu'à l'*exécution*, soit plus tard. Avec le modèle d'instanciation de `template` à la compilation du C++, il n'est donc pas possible de mêler `virtual` et `template` pour une méthode.

Attention à ne pas non plus confondre méthodes polymorphes/virtuelles et surcharge de fonctions (*overloading*) !

a. Ou des instanciations explicites de cette méthode paramétrée.

Best-of:
- Les templates sont résolus dynamiquement contrairement à l'appel d'une méthode virtuel.
- [. . .] Or `template` ne type qu'à l'exécution.
- [. . .] Cela serai une horreur si fait.
- Car Bjarne Stroustrup était flemmard.
- Il faut faire un choix dans la vie. Bjarne l'a fait pour nous.

3. **Bonus.** Écrivez un bout de code C++ qui calcule statiquement (à la compilation, donc) la valeur 10! (bien entendu, l'affichage du résultat sera dynamique, puisqu'il faudra quand même exécuter le programme pour ce faire).

Correction: Il s'agissait bien entendu de calculer 10!, à savoir *factorielle* 10, et non de « calculer » l'*entier* 10 (comme je l'ai vu sur trop de copies), car cela n'aurait pas eu beaucoup de sens.
Pour ce faire, on utilise la métaprogrammation via les templates du C++ :

```
#include <iostream>

template <unsigned u>
struct fact
{
  static const unsigned res = u * fact<u - 1>::res;
};

template <>
struct fact<0>
{
  static const unsigned res = 1;
};

int main ()
{
  std::cout << fact<10>::res << std::endl;
}
```

La valeur 10! est alors effectivement calculée à la compilation (et affichée à l'exécution).

Best-of:

–
```
#include <iostream>

int
main ()
{
  std::cout << 5 + 5 << std::endl;
}
```

–
```
#define valeur 5+5
printf("%d", valeur);
```

–
```
int main () { return 10; }
```