Correction du partiel CMP1

EPITA – Promo 2014
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices, ordinateurs, tablettes et téléphones interdits.

Janvier 2012 (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 (entier) est recommandée.

1 Le retour du Logo

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

```
⟨instrs⟩ ::= ⟨instr⟩+

⟨instr⟩  ::= "forward" ⟨exp⟩
           | "left" ⟨exp⟩
           | "right" ⟨exp⟩

           # Loop.
           | "repeat" ⟨exp⟩ "[" ⟨instrs⟩ "]"

           # Subprogram definition.
           | "to" "id" ⟨args⟩ ⟨instrs⟩ "end"

           # Subprogram call.
           | "id" ⟨exps⟩

⟨exps⟩   ::= ⟨exp⟩*

⟨exp⟩    ::= "num"
           | ":" "id"

⟨args⟩   ::= ⟨arg⟩*

⟨arg⟩    ::= ":" "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 originel permet notamment 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). Il permet également la définition de sous-programmes (procédures) à l'aide du mot-clef `to`, qui sont exécutés lorsque leur nom est utilisé comme instruction.

1. On souhaite reconnaître ce langage à l'aide d'un scanner et d'un parser engendrés par Flex et Bison respectivement. Dressez une liste des tokens qui seront produits par le scanner et utilisés par le parser.

Correction:

ID	(id)	FORWARD	("forward")
NUM	(num)	LEFT	("left")
COLON	(":")	RIGHT	("right")
LBRACK	("[")	REPEAT	("repeat")
RBRACK	("]")	TO	("to")
END	("end")	EOF	(end of file)

Best-of:

– *Cette question a donné lieu a de jolis mélanges d'anglais et de français, dans la plus pure tradition épitéenne :*

ACC_OP, ACC_CL	LEFT_C, RIGHT_C
ACCOLADELEFT, ACCOLADERIGHT	LEFTPARANT, RIGHTPARANT
CROCHET_OPEN, CROCHET_CLOSE	OPEN_CROCHET, CLOSE_CROCHET
CROL, CROR	OPENACCO, CLOSEACCO
LACC, RACC	TOKEN_LEFT_CRO, TOKEN_RIGHT_CRO
LCRO, RCRO	TOKEN_OPEN_CROCH, TOKEN_CLOSE_CROCH
LEFT_CR, RIGHT_CR	

– *Ainsi que d'autres propositions intéressantes :*

L_CROUCH, R_CROUCH	OPENHOOK, CLOSEHOOK
LEFT_HOOK, RIGHT_HOOK	TOKEN_LHOOK, TOKEN_RHOOK
LEFTHOOK, RIGHTHOOK	

2. Comment faire comprendre à votre scanner Flex que l'entrée '`forward`' doit produire un token FORWARD et non un token ID (correspondant au symbole '`forward`') ?

Correction: Les règles Flex générant respectivement le token FORWARD 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:

– Il faut mettre le `forward` à gauche.
– Il faut ajouter des règles pour désambiguïser ces cas (à l'aide des shift/reduce par exemple). Il reconnaîtra le mot-clé de toute façon.

3. Pourquoi est-ce que votre scanner Flex générera à partir de l'entrée Tiger '`to`' un seul token TO, et non un token ID représentant l'identifiant '`t`' suivi d'un autre token ID représentant l'identifiant '`o`' ?

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 que la seule présence ou absence d'espace entre les deux tokens suffit pour désambiguïser.

Best-of:
- Le scanner possède par défaut un séparateur de mot : l'espace.
- Flex est un scanner bien conçu donc en voyant le '`t`' il va regarder le caractère suivant qui est un '`o`' donc il va shifter et retourner un token TO plutôt que 2 ID. Flex cherchera le plus long mot possible. (*Ouf, la bonne réponse a fini par arriver !*)
- Le scanner Flex produira à partir de l'entrée un seul token TO car lors de l'ajout de fonctionnalité, il y a une étape de liaison des noms.

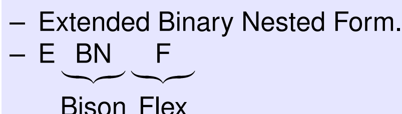
4. Cette grammaire présente-t-elle une (des) ambiguïté(s) ? Si oui, donner un exemple de mot du langage engendré par celle-ci, donnant lieu à plus d'une dérivation.

Correction: Non, aucune ambiguïté.
Plusieurs copies indiquent que la séquence de tokens « : ID » peut donner lieu à plusieurs interprétations, puisque qu'elle peut se réduire en `exp` ou en `arg` ; cela rendrait donc la grammaire ambiguë. Il n'en est rien, puisque ces deux non terminaux sont utilisés à des endroits différents dans la grammaire.

Best-of:
- Le token '`id`' est très ambigüe.

5. Que signifie EBNF ?

Correction: Extended Backus-Naur Form.
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.

Best-of:
- Extended Binary Nested Form.
- 

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

Correction: Bison ne sait pas traiter directement une grammaire EBNF comme celle de l'énoncé. Il nous faut tout d'abord la traduire en BNF avant de l'adapter pour Bison. Les règles qui posent problème sont les productions de `instrs`, `exps` et `args`.
Nous pouvons les réécrire ainsi pour les rendre immédiatement traduisibles dans Bison :

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

⟨exps⟩   ::= ⟨exps⟩ ⟨exp⟩ |

⟨args⟩   ::= ⟨args⟩ ⟨arg⟩ |
```

Best-of:
- On peut faire l'ajout d'une fonctionnalité « CONST » (. . .) De plus on peut ajouter des « start conditions » de lex/flex.

7. On souhaiterait que le parser sache faire un peu de reprise sur erreur, afin que celui-ci produise des messages plus fournis (ne se limitant pas à la première erreur rencontrée). On utilise pour cela le token spécial '`error`' produit en cas d'erreur.

   Quelle(s) règle(s) ajouteriez-vous à un fichier d'entrée Bison issu de la grammaire modifiée de la question 6 afin d'ajouter cette reprise sur erreur ?

   Correction:
   (Note : il aurait été plus logique que cette question soit située *après* la question 8, même si son emplacement actuel ne change pas la réponse attendue, avec ou sans actions sémantiques.)

   Deux bons emplacements pour effectuer la reprise sur erreur sont le corps d'une boucle (instruction '`repeat`') et la définition d'un sous-programme (instruction '`to`'), car ils sont encadrés (respectivement par '`[`'/'`]`' et '`to`'/`end`).
   Ainsi, si la grammaire Bison comportait les règles ci-dessous :

```
instr :
  "repeat" exp "[" instrs "]" { $$ = new Loop ($2, $4); }
| "to" "id" args instrs "end" { $$ = new Subprogram ($2, $3, $4); }
```

   une version augmentée avec de la reprise sur erreur serait :

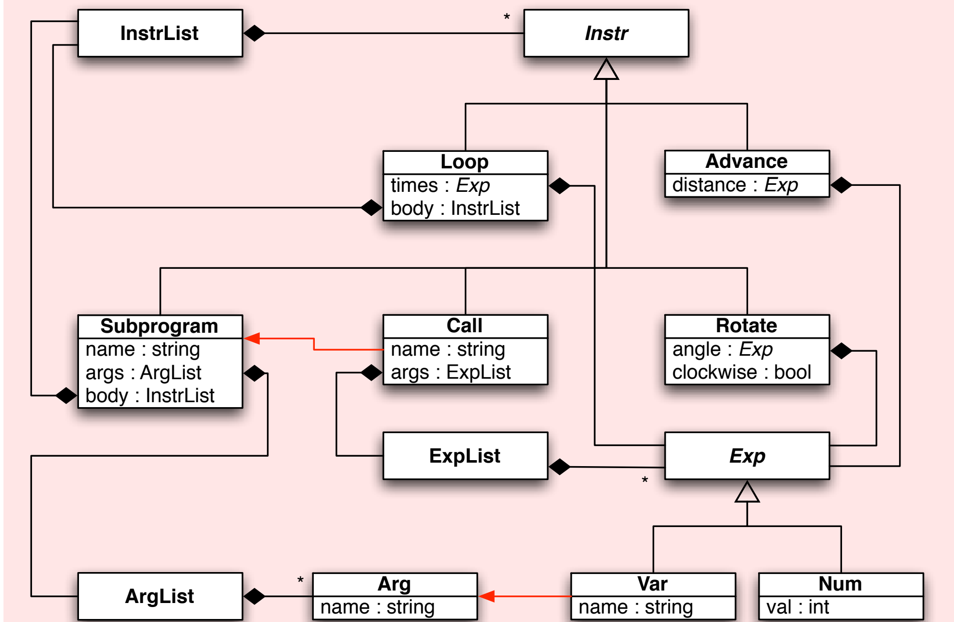
```
instr :
  "repeat" exp "[" instrs "]" { $$ = new Loop ($2, $4); }
| "repeat" exp "[" error "]"  { $$ = new Loop ($2, new InstrList); }
| "to" "id" args instrs "end" { $$ = new Subprogram ($2, $3, $4); }
| "to" "id" error "end"       { $$ = new Subprogram ($2,
                                                     new ArgList,
                                                     new InstrList); }
```

   `InstrList` et `ArgList` sont des nœuds qui représentent des listes d'instructions et d'arguments respectivement. Ce n'était pas attendu dans la réponse, mais signalons pour information que l'on devra éventuellement ajouter des directives `%destructor` pour libérer correctement la mémoire associée aux symboles défaussés lors de la reprise sur erreur. Enfin, certaines copies parlent à tort de `yyerror` ou de `yy::parser::error` : la question portait sur la reprise sur erreur (à l'aide du token '`error`') et non sur la gestion des erreurs.

   Best-of:
   – J'ajouterais une règle qui, dans tous les cas autres que ceux de la grammaire, renverrait une erreur.
   – Il faut ajouter `yylex` et `yyval`.

8. Les arbres de syntaxe abstraite ou Abstract Syntax Trees (ASTs) produits par le parser engendré 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, soit sous forme de code C++. Vous pouvez indiquer les données (attributs) des dites classes, mais il est inutile de mentionner leurs méthodes.

Correction:



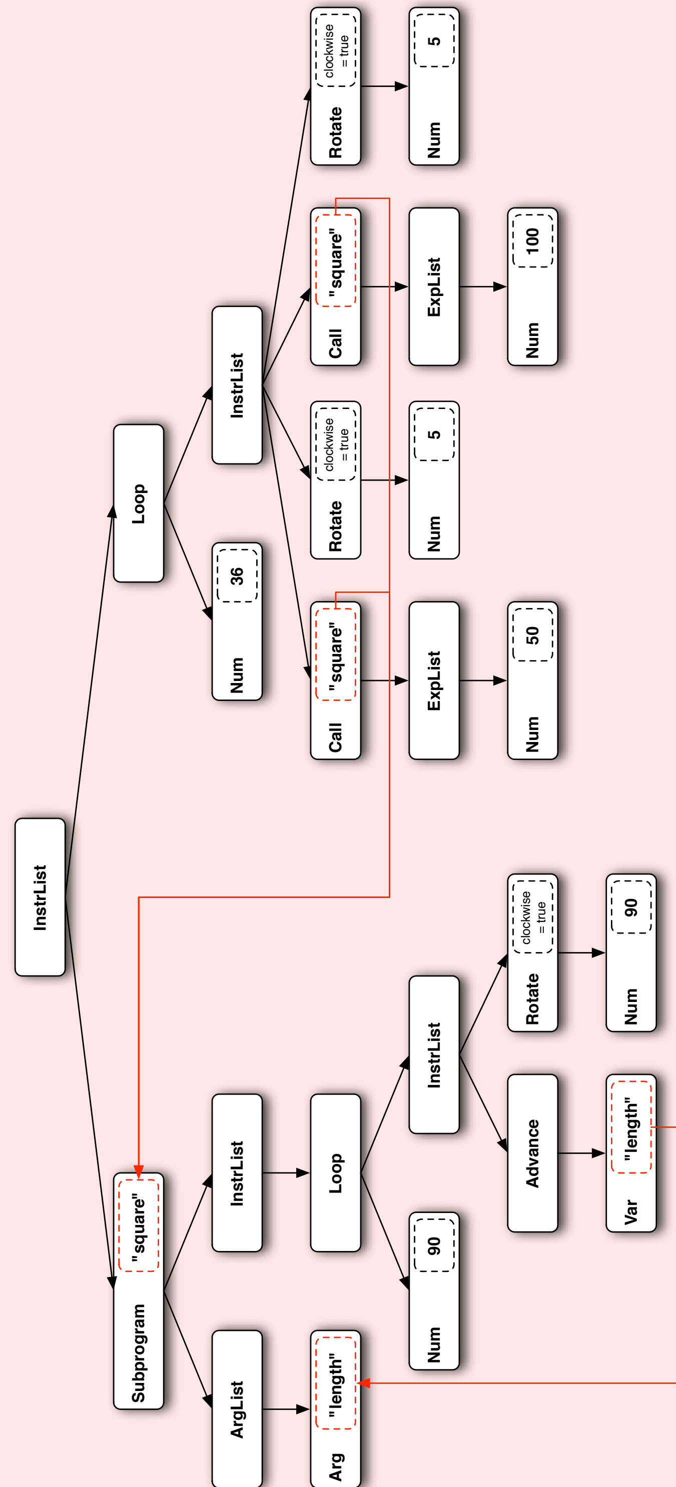
Pour information, le diagramme ci-dessus montre en plus (ce n'était pas demandé) les liaisons entre sites de définition de noms (resp. `figs/Subprogram` et `Arg`) et les sites d'utilisation de ces noms (resp. `Call` et `Var`), à l'aide de flèches rouges.

9. Voici un programme Logo qui dessine une forme de rosace :

```
to square :length
repeat 4 [ forward :length right 90 ]
end
repeat 36 [ square 50 right 5 square 100 right 5 ]
```

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 8 :

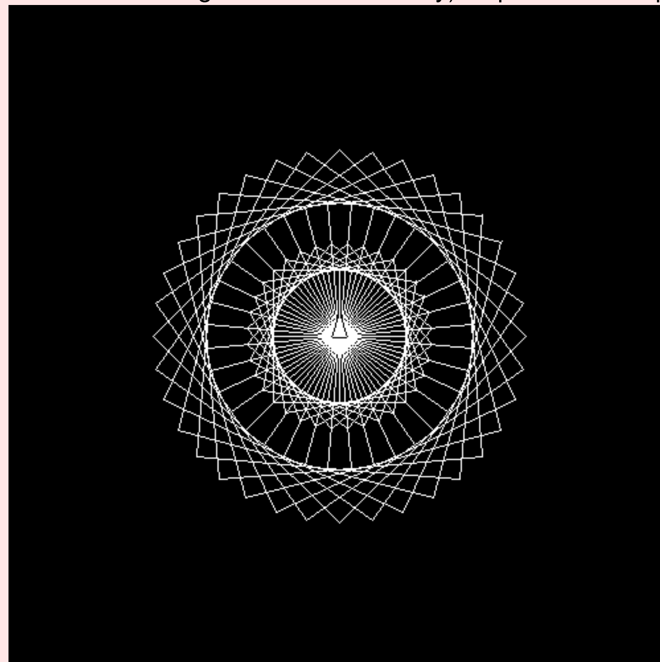


Comme dans la réponse précédente, ce diagramme montre également, à l'aide de flèches rouges, les informations de liaison entre définitions et utilisations de noms (qui n'étaient pas demandées dans la question).

Correction: (suite) Une version textuelle :

```
InstrList
  (Subprogram ("square",
               ArgsList (Arg ("length")),
               InstrList
                 (Loop (Num (4),
                        InstrList (Advance (Var ("length")),
                                   Rotate (true, Num (90)))))),
   Loop (Num (36),
         InstrList (Call ("square", ExpList (Num (50))),
                    Rotate (true, Num (5)),
                    Call ("square", ExpList (Num (100))),
                    Rotate (true, Num (5)))))
```

Attention à ne pas donner l'arbre de dérivation (*parse tree*) au lieu de l'AST. En particulier, on ne devait plus trouver trace de '[', ']', ou ':' dans cet arbre. Dans certaines copies, la réponse donnée était la représentation graphique produite par l'*exécution* du programme Logo ; ce n'était pas ce qui était demandé. Cela étant dit, et pour information, voici ce que dessine `ucblogo` (une implémentation de Logo de l'UC Berkeley) à partir de ce programme :



10. Voici un extrait du code de la syntaxe abstraite de notre langage Logo.

```
class Ast
{
public:
  virtual ~Ast () {};
  virtual void accept (Visitor& v) const = 0;
};

class Instr : public Ast
{
};
```

```
class Subprogram : public Instr
{
public:
  Subprogram (const std::string& name, const ArgList* args,
              const InstrList* body)
    : name_ (name), args_ (args), body_ (body)
  {
  }
  virtual ~Subprogram () { delete args_; delete body_; }

  std::string      name_get() { return name_; } const
  const ArgList*   args_get() { return args_; } const
  const InstrList* body_get() { return body_; } const

  virtual void accept (Visitor& v) const { v (*this); }

private:
  std::string name_;
  const ArgList* args_;
  const InstrList* body_;
};

// ...
```

On souhaite implémenter un pretty-printer pour cette syntaxe abstraite grâce à un visiteur, affichant récursivement un AST. Celui-ci sera implémenté sous la forme d'une classe `PrettyPrinter`, dérivée de la classe abstraite `Visitor`, dont un extrait est donné ci-dessous :

```
struct Visitor
{
  virtual void operator() (const Right& e) = 0;
  virtual void operator() (const Left& e) = 0;
  virtual void operator() (const Subprogram& e) = 0;
  // ...
};
```

Écrivez la classe `PrettyPrinter`, en limitant les méthodes de visite ('`operator()`') à la seule classe `Subprogram` donnée ci-avant (on souhaite cependant voir le reste de la classe).

On rappelle qu'un visiteur permet l'encapsulation d'un traitement spécialisé pour toutes les classes concrètes d'une hiérarchie de classes (en l'occurrence, celle de notre syntaxe abstraite) *et* des données associées ; tenez-en compte dans votre réponse ! (Note : le code à écrire n'est pas très long ; environ une quinzaine de lignes.)

Correction: Une première version serait :

```
class PrettyPrinter : public Visitor
{
public:
  PrettyPrinter (std::ostream& ostr) : ostr_ (ostr) {}

  virtual void operator() (const Subprogram& e)
  {
    ostr_ << "to " << e.name_get () << " ";
    e.args_get ()->accept (*this);
    ostr_ << " ";
    e.body_get ()->accept (*this);
  }

  // Other 'operator()' methods, not shown here.
  // ...

private:
  std::ostream& ostr_;
};
```

Ici, on fait du flux de sortie (`ostr_`) un membre du visiteur, car c'est une donnée du traitement (l'affichage de l'AST).
Cependant, l'alternance d'appels à '<<' et à `accept()` est à la fois disgracieuse et verbeuse. Pour améliorer la solution précédente, on peut se donner le sucre syntaxique ci-dessous :

```
std::ostream&
operator<< (std::ostream& ostr, const Ast& tree)
{
  PrettyPrinter printer (ostr);
  tree.accept (printer);
  return ostr;
}
```

qui permet de simplifier l'écriture de la routine d'affichage d'instances de '`Subprogram`' :

```
  virtual void operator() (const Subprogram& e)
  {
    ostr << "to " << e.name << " " << *e.args << " " << *e.body;
  }
```

2 De la virtualité et de la destruction en C++

1. Qu'appelle-t-on une fonction membre virtuelle pure en C++ (également connue sous le nom de « méthode (polymorphe) abstraite ») ?

   Correction: Une méthode dépourvue d'implémentation (abstraite) et redéfinissable dans une classe dérivée. La sélection de la méthode effectivement appelée se fait en fonction du type exact (dynamique) de la cible (argument « zéro », `*this` en C++).

2. Pourquoi ne doit-on pas appeler de méthodes virtuelles d'un objet lors de la construction de celui-ci ?

   Correction: Rappelons que la construction d'un objet se fait « de haut en bas ». Le constructeur de sa classe appelle celui (ceux) de sa (ses) classe(s) parente(s), etc. ; la partie de l'objet correspondant à la classe parente la plus haute est initialisée, puis celles de ses sous-classes, et ainsi de suite jusqu'à la classe de l'objet. L'objet n'est complètement construit que lorsque toutes ses parties ont été construites.
   Or si une méthode virtuelle est appelée par le constructeur d'une des classes parentes de l'objet, il est possible que l'implémentation effective de celle-ci appartienne à une classe située plus bas dans la hiérarchie, et encore non « enregistrée » dans le mécanisme de dispatch (en C++, cela se traduirait par une table de fonctions virtuelles potentiellement incorrecte).

   Best-of:
   – Car il y a compilation séparée.

3. Lors de la situation de la question précédente, les compilateurs C++ déclenchent habituellement une erreur. Par exemple, avec le code ci-dessous :

```
struct A
{
  A () { m (); }
  virtual void m () = 0;
};

struct B : A
{
  virtual void m () {};
};

int main ()
{
  A* b = new B;
  b->m ();
  delete b;
}
```

   on obtient l'erreur suivante à l'édition de liens avec g++ 4.4 :

```
a.cc: In constructor 'A::A()':
a.cc:3: warning: abstract virtual 'virtual void A::m()' called from
  constructor
/tmp/ccfvyPSG.o: In function `A::A()':
a.cc:(.text._ZN1AC2Ev[A::A()]+0x1f): undefined reference to `A::m()'
collect2: ld returned 1 exit status
```

Malgré cela, le code ci-dessous compile bel et bien avec le même compilateur :

```
1  struct A
2  {
3    A () { f (); }
4    void f () { m (); }
5    virtual void m () = 0;
6  };
7
8  struct B : A
9  {
10   virtual void m () {};
11 };
12
13 int main ()
14 {
15   A* b = new B;
16   b->m ();
17   delete b;
18 }
```

Il affiche cependant l'erreur suivante à l'exécution :

```
pure virtual method called
terminate called without an active exception
```

(a) Expliquez ce qui s'est passé lors de l'exécution de ce programme.

Correction: Version courte : lors de la construction de `b`, le constructeur `B::B()` appelle `A::A()` qui appelle la méthode virtuelle `m()` via `f()` ; or cette méthode n'a pas encore été installée dans le système de dispatch de `b` à ce moment (cela n'aura lieu qu'après la fin de `B::B()`) ; dans l'intervalle, `A::A()` a installé une sentinelle, qui est appelée au lieu de la routine `B::m()` attendue, déclenchant l'erreur figurant dans l'énoncé.

Correction: Version longue : L'instanciation d'un objet de type `B` (ligne 15) provoque l'appel du constructeur sans arguments de cette classe (`B::B()`), qui est présent, même si le code ne le montre pas. En effet, celui-ci est ajouté implicitement par le compilateur, puisque `B` ne déclare aucun constructeur explicitement. Comme tout constructeur, `B::B()` appelle tous les constructeurs de ses classes parentes, en l'occurrence `A::A()` (ligne 3), et ce *avant* d'initialiser le contenu de l'objet (`b` dans `main()`) correspondant à `B` : cela inclut l'enregistrement de la méthode virtuelle `B::m()` dans le mécanisme de dispatch de `b`. `A::A()` initialise « l'étage » `A` dans l'objet `b` en cours de construction, qui comporte l'installation d'une routine d'erreur (une sentinelle) dans l'emplacement correspondant à `A::m()` dans la table de fonctions virtuelles (*vtable*) de l'objet `b`. Cette routine est une sécurité, chargée d'arrêter le programme si jamais `A::m()` était effectivement appelée (ce qui n'arrive normalement pas, sauf dans de rares cas comme celui qui est mis en évidence dans cette question). `A::m()` appelle ensuite `f()`, qui est une méthode non virtuelle, et dont l'exécution ne pose a priori aucun problème depuis un constructeur. Sauf que cette méthode déclenche elle-même un appel à la méthode virtuelle `m()`, dont l'adresse est censée être contenue dans la table de fonctions virtuelles de `b` ; or comme dit précédemment, celle-ci n'est pas complètement installée à ce stade du programme, puisque `B::B()` n'a pas terminé son initialisation. Le *slot* contenant l'adresse de `m()` ne dispose donc pas encore de la « bonne » adresse (celle de `B::m()`) ; à la place figure l'adresse de la routine provoquant l'erreur mentionnée plus haut. C'est donc celle-ci qui est appelée au lieu de `B::m()`.

Remarque : dans le premier message d'erreur, le compilateur ne donne qu'un avertissement au sujet de l'appel de la méthode `m()` depuis le constructeur `A::A()`, et ne cherche pas à mettre en place le mécanisme de sélection dynamique (*dynamic dispatch*) pour `m()` ; à la place, il lie directement cet appel à `A::m()`, mais comme celle-ci est abstraite (pas d'implémentation), l'éditeur de liens est incapable de la trouver et donne une erreur.
Tous les compilateurs ne fonctionnent pas ainsi ; par exemple, `clang` 3.0 compile le premier code avec un avertissement, mais sans erreur à l'édition de liens ; le programme produit génère bien sûr une erreur à l'exécution (`pure virtual method called`) ; ici le compilateur a mis en place un dispatch dynamique pour `m`, mais celui-ci ne pouvait bien entendu pas aboutir.

(b) À votre avis, pourquoi le compilateur n'a pas été capable de prévenir que le programme était mal formé (*ill-formed*) ?

Correction: Dans le premier programme, `m()` est appelée directement depuis `A::A()` : ce cas pathologique est facilement détectable par le compilateur, qui peut immédiatement vérifier que *tous* les appels directs de fonctions effectués depuis un constructeur ne sont pas virtuels. Comme il n'est jamais légitime d'appeler une méthode virtuelle depuis un constructeur, tout appel à une méthode virtuelle est une erreur (même s'il pourrait l'être sous une certaine condition dynamique que le compilateur ne peut pas évaluer).

Dans le second programme, l'appel à `m()` est indirect : `A::A()` appelle `f()` (non virtuelle) qui appelle `m()` (virtuelle). On constate que l'heuristique de détection d'appels virtuels depuis un constructeur utilisée par `g++` est assez simple : elle se limite aux appels directement effectués depuis le constructeur lui-même. Par conséquent, `g++` ne « voit » pas l'appel à `m()`, indirect.
Ce comportement du compilateur est justifié : une vérification complète nécessiterait une analyse statique en profondeur (et donc coûteuse en temps de compilation), qui pourrait même se révéler insuffisante. En effet, si l'implémentation de `f()` était placée dans une autre unité de compilation, elle serait inaccessible au compilateur lors de l'analyse de `A`. Par ailleurs, il se peut que `f()` n'appelle pas systématiquement `m()` (il y a peut-être un `if` jamais vérifié dans le cas de l'appel depuis le constructeur). Il n'y a donc pas de moyen statique de vérifier si toutes les exécutions de `f()` depuis le constructeur sont saines.

4. Considérons à présent le code C++ suivant :

```
1  struct C
2  {
3    virtual ~C () = 0;
4  };
5
6  struct D : C
7  {
8  };
9
10 int
11 main ()
12 {
13   D d;
14 }
```

Quelle peut être l'utilité de la ligne 3 ?

Correction: Le fait que le destructeur de la classe `C` (`~C`) soit précédé du mot-clef `virtual` enclenche le mécanisme de dispatch dynamique pour ce dernier. Cela signifie qu'une instance de `C` ou d'une classe dérivée de `C` (telle `D`), lors de sa destruction, appellera le destructeur de son type exact (le type dynamique de l'objet). Cela est particulièrement important si cet objet doit libérer des ressources (mémoire, descripteurs de fichiers, verrous, etc.) via le destructeur de son type exact.
De plus `~C` est suivi de '`= 0`'. Cela signifie que ce destructeur est abstrait (ou virtuel pur) : il ne possède pas d'implémentation. Cela rend donc automatiquement la classe `C` abstraite (non instanciable).
Notons que lorsqu'une classe possède une méthode abstraite, ses sous classes concrètes doivent fournir une implémentation de cette méthode : mais ici, cela n'est pas possible ! En effet, une classe dérivée de `C`, telle `D`, ne peut contenir un destructeur de `C`. Le seul destructeur que `D` possède (explicitement ou implicitement) est le sien propre (`D::~D`).

Best-of:
- L'utilité de `virtual ~C () = 0` est de mettre à Null le pointeur après destruction.
- À déconstruire la structure `C`.

5. Le code de la question 4 compile bien, mais provoque une erreur à l'édition de liens. Le message produit par g++ 4.4 est le suivant :

```
/tmp/ccQdx4dr.o: In function 'D::~D()':
b.cc:(.text._ZN1DD1Ev[D::~D()]+0x1f): undefined reference to 'C::~C()'
/tmp/ccQdx4dr.o: In function 'D::~D()':
b.cc:(.text._ZN1DD0Ev[D::~D()]+0x1f): undefined reference to 'C::~C()'
collect2: ld returned 1 exit status
```

À quoi est due cette erreur ? (« Le destructeur de `C` est absent » n'est pas une réponse suffisante.)

Correction: À la fin de la fonction `main()` (ligne 14), les objets temporaires (locaux) sont détruits. Ici, il s'agit de `d` (ligne 13). Le destructeur de cet objet est appelé. Rappelons que même si le code de `D` (lignes 6–8) ne comporte pas de destructeur explicite, celui-ci existe quand même. En effet, le compilateur ajoute dans le code produit une version implicite du destructeur, qui ne contient aucune instruction. Ce destructeur `D::~D` implicite n'effectue aucune action relative à `D`, mais *comme tout destructeur*, il appelle les destructeurs des classes parentes, en l'occurrence, `C::~C` (ligne 3). Or ce dernier n'existe pas, puisqu'il est abstrait (virtuel pur). Ce problème se traduit par une erreur à l'édition de liens. Lors de la compilation, le compilateur a validé l'existence du nom `C::~C` (ce qui a rendu la compilation possible), mais ne lui a associé aucune adresse (ce qui a déclenché l'erreur du linker).
Beaucoup de copies expliquent que l'erreur provient de l'absence de destructeur dans `D`. Le message d'erreur était pourtant explicite et aurait dû mettre leurs auteurs sur la voie : le problème est effectivement constaté à l'intérieur de `D::~D`, mais concerne `C::~C`. Le premier ne trouve pas le second lors de l'appel récursif (en cascade) des destructeurs dont il est fait mention plus haut.

Best-of:
- Il n'y a pas de constructeur, donc une erreur est affichée à cause du destructeur.
- Il n'y a pas de deconstructeur virtual `~C` dans `D`.

6. Même si cela paraît contre nature, le langage C++ autorise la définition d'une implémentation pour une fonction membre virtuelle pure. Dans le cas d'un destructeur virtuel pur, le standard ISO C++ précise qu'une telle définition est même indispensable, si l'on veut être en mesure de pouvoir instancier la classe contenant ce destructeur ou l'une de ses classes dérivées.

   Fort de cette information, qu'ajouteriez/modifieriez-vous dans le code de la question 4 pour le rendre propre à la compilation et à l'exécution ?

Correction: Il suffit de donner une implémentation vide du destructeur en question :

```
C::~C () {}
```

Malgré son caractère « abstrait » (virtuel pur), celui-ci sera effectivement appelé lors de la destruction d'instances de `D` par le destructeur de ce dernier (destruction récursive de bas en haut). Pour des question de syntaxe, il est impossible de définir l'implémentation de `C::~C` dans la déclaration de la classe `C`. L'utilisation de '`= 0`' interdit en effet l'ajout d'une définition à la déclaration du destructeur. Note : beaucoup de copies suggèrent d'ajouter un destructeur à `D`, mais comme dit précédemment, là n'est pas le problème : celui-ci existe déjà, ajouté implicitement par le compilateur. D'autres proposent carrément d'ajouter un destructeur (explicite) de `C` dans la classe... `D` !

Best-of:
– Le standard ISO C++ est une blague, une mauvaise blague.