Composition d'Informatique
Les Principes des Langages de Programmation (INF 321)
Promotion 2012

Sujet et correction proposée par Eric Goubault

12 juillet 2013

Le problème se compose de quatre parties, le tout noté sur 20, et d'une cinquième partie, optionnelle, notée sur 4. Le barême donné est indicatif. Les parties 3 et 4 sont indépendantes de 1 et 2. La partie 5 est indépendante des autres parties. On attachera une grande importance à la clarté, à la précision et à la concision de la rédaction et des explications.

On considère dans ce problème la modélisation et l'implémentation en JAVA de la calculatrice "trois opérations" de la figure 1. Dans les parties 3, 4 et 5, on considère les touches supplémentaires *prgm* et *run* qui en font une calculatrice programmable

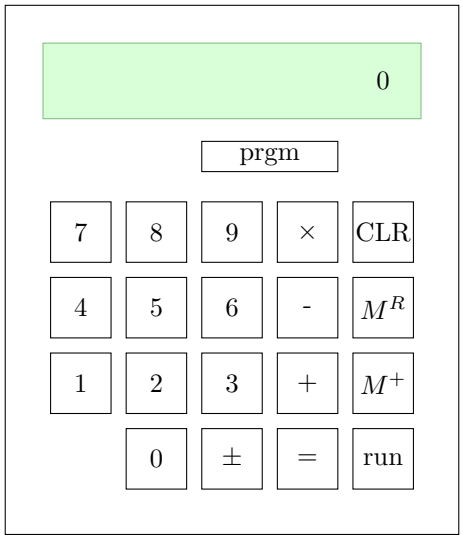

Figure 1: Une simple calculatrice trois opérations

Celle-ci comporte un clavier numérique, permettant d'entrer des nombres *entiers* (donc dans $\mathbb{Z}$) à l'écran, ainsi que des touches permettant d'effectuer des opérations simples. Elle a également une *mémoire*, c'est-à-dire un registre pouvant stocker un entier, et un *afficheur*. Les touches $+$, $-$ et $\times$ effectuent l'addition, la soustraction et la multiplication de deux nombres. Une touche $\pm$ change le signe du nombre à l'affichage. La touche $=$ permet d'obtenir le résultat d'un calcul en cours. Une touche CLR efface le nombre courant, à l'affichage et revient à l'état initial de la calculatrice (plus rien en mémoire). Enfin une touche M^+ permet de stocker la valeur affichée vers la mémoire interne, et M^R permet de rappeler celle-ci à l'écran.

Ainsi, un utilisateur rentrant la combinaison de touches suivantes (auquelle on fera référence par la suite comme étant la combinaison C), à partir de l'état initial de la calculatrice:

$$10\ M^+\ +\ 6\ \pm\ =\ \times\ M^R\ =$$

obtiendra à l'écran la valeur 40 ($(10-6)\times 10$).

1 Types de données

La calculatrice a un état courant qui comporte le nombre à l'affichage, que l'on nomme V, et la valeur courante de la mémoire M. Mais pour calculer, cette calculatrice a besoin d'un registre "caché", que l'on appelera par la suite accumulateur, noté A et un "flag" F donnant l'opération courante (égal à $+$, $\times$, $-$ ou *nop* si on n'a aucune opération en cours). En effet, quand l'utilisateur démarre C en tapant le nombre 10, celui-ci est visible et est donc stocké dans V. Taper M^+ le stocke également dans M. Le fait d'appuyer sur $+$ n'effectue pas encore l'opération, celle-ci est simplement, pour l'heure, stockée dans F. Quand l'utilisateur rentre le chiffre 6, le nombre 10 disparaît de l'affichage, mais est conservé dans l'accumulateur A, 6 étant maintenant stocké dans V. Le symbole $\pm$ change dans V la valeur 6 en -6. Enfin, presser la touche $=$ active le calcul de l'opération stockée dans F ($+$ ici) sur les nombres stockés en A et V, c'est-à-dire 10 et -6 respectivement, aboutissant à 4, nouvelle valeur à l'affichage, dans V, l'accumulateur étant alors effacé (égal à zéro).

On résume l'action de la combinaison de touches C sur l'état de la calculatrice, identifié au quadruplet (A, V, M, F):

Touche/Etat	A	V	M	F
	0	0	0	*nop*
10	0	10	0	*nop*
M^+	0	10	10	*nop*
$+$	0	10	10	$+$
6	10	6	10	$+$
$\pm$	10	-6	10	$+$
$=$	0	4	10	*nop*
$\times$	0	4	10	$\times$
M^R	4	10	10	$\times$
$=$	0	40	10	*nop*

(1) (1 point) Définissez un type de données JAVA, le mieux adapté possible et que l'on nommera `Flag`, pour représenter F

(2) (2 points) Définissez une classe JAVA, que l'on nommera `State`, pour représenter l'état de la calculatrice (le quadruplet (A, V, M, F))

Correction

(1)
```
public enum Flag { Plus, Times, Minus, Nop };
```

```
public class State {
  int A;
  int V;
  int M;
  Flag F;
}
```

2 Sémantique et interpréteur

On notera $\mathcal{D} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \{p, m, t, n\}$ le domaine mathématique[1] représentant l'état de la calculatrice: $(A, V, M, F) \in \mathcal{D}$ est le quadruplet où A est la valeur de l'accumulateur, V est la valeur de l'affichage, M est la valeur de la mémoire et où F donne l'opération courante, p pour l'addition, m pour la soustraction, t pour la multiplication et n pour aucune opération courante.

(3) (2 points) Donner la sémantique $[\![T]\!]\rho$ de chaque action T (nombre rentré sur le clavier numérique, M^R, M^+, CLR, $+$, $-$, $\times$, $=$, $\pm$) évaluée dans l'environnement $\rho \in \mathcal{D}$. On fera attention au fait qu'une séquence telle:

$$10 + 20 + 30 =$$

est évaluée à 60, c'est-à-dire que si F n'est pas *nop*, toute opération nouvelle de type $+$, $-$ ou $*$ force le calcul courant à être effectué. De plus, la calculatrice ne gère pas les priorités d'opérations arithmétiques, donc $10 + 2 * 4 =$ donne 48 ($(10+2)\times 4$) et non 18 ($10+(2\times 4)$).

(4) (5 points)

(a) (2 points) On écrira successivement les méthodes *dynamiques* de `State`, qui modifient *en place* l'état courant:

```
void mr();
void mp();
void clr();
void plus();
void minus();
void times();
void equal();
void pm();
```

correspondant respectivement à l'action de la touche M^R, M^+, CLR, $+$, $-$, $\times$, $=$ et $\pm$

(b) (2 points) Ecrire une méthode *dynamique* Java de la classe `State`:

```
void calc();
```

permettant d'interpréter l'action des touches de la calculatrice: `calc` simule donc son comportement, en changeant en place l'état courant.

On identifiera précisément dans le code, à l'aide de commentaires Java (`//`) les règles de la sémantique donnée à la question précédente que l'on implémente.

On supposera donnée une méthode `static Touch touch()` qui:

* bloque si aucune touche n'a été tapée
* ou renvoie l'instruction qui vient d'être tapée sur le clavier dans le champ `i` de `Touch` (défini plus bas) et *null* dans le champ x,
* ou renvoie un nombre dans x, si l'utilisateur n'a pas tapé sur une instruction mais sur une suite de touches numériques, suivi d'une instruction, qu'il renvoie également dans i

On supposera que la classe `Touch` est définie comme suit:

```
public class Ent {
  public int val;
}

public enum Action { mr, mp, clr, plus, moins, mult, egal, pm };
```

[1] En fait, l'implémentation JAVA, ainsi que la calculatrice, ne considèrent que les entiers accessibles par une représentation `int` sur 32 bits, classique.

```
public class Touch {
  public Ent x;
  public Action i;
}
```

`Action` est le type de données correspondant à toutes les touches sur lesquelles on peut appuyer (respectivement M^R, M^+, CLR, $+$, $-$, $\times$, $=$, $\pm$), à part la partie numérique du clavier.

(c) (1 point) On souhaite éviter que si l'on tape la suite de touches:

$$10 \; + \; 20 \; = \; + \; =$$

on obtienne 30, mais plutôt un avertissement à l'utilisateur lui disant que le deuxième plus manque d'un argument. Quel mécanisme devrait-on utiliser en Java pour signaler proprement à l'utilisateur ces cas d'erreur? Indiquer le code correspondant à rajouter à l'interpréteur de la calculatrice

Correction

(3) On écrit:

$$\begin{array}{lcl}
[\![n]\!](A,V,M,F) & = & (V,n,M,F) \\
[\![\pm]\!](A,V,M,F) & = & (A,-V,M,F) \\
[\![M^R]\!](A,V,M,F) & = & (V,M,M,F) \\
[\![M^+]\!](A,V,M,F) & = & (A,V,V,F) \\
[\![clr]\!](A,V,M,F) & = & (0,0,0,nop) \\
[\![=]\!](A,V,M,F) & = & \begin{cases} (0,A+V,M,nop) & \text{si } F = + \\ (0,A-V,M,nop) & \text{si } F = - \\ (0,AV,M,nop) & \text{si } F = \times \\ (A,V,M,nop) & \text{si } F = nop \end{cases} \\
[\![+]\!](A,V,M,F) & = & (\pi_{13}([\![=]\!](A,V,M,F)),+) \\
[\![-]\!](A,V,M,F) & = & (\pi_{13}([\![=]\!](A,V,M,F)),-) \\
[\![\times]\!](A,V,M,F) & = & (\pi_{13}([\![=]\!](A,V,M,F)),\times)
\end{array}$$

où $\pi_{13}(A,V,M,F) = (A,V,M)$.

(4) (a)
```
public class State {
  ...

  void mr() {
    A = V;
    V = M;
  }

  void mp() {
    M = V;
  }

  void clr() {
    A = 0;
    V = 0;
    M = 0;
    F = Nop;
  }
```

```
  void plus() {
    equal();
    F = Plus;
  }

  void minus() {
    equal();
    F = Minus;
  }

  void times() {
    equal();
    F = Times;
  }

  void equal() {
    switch F {
      case Plus: V = A+V;
        break;
      case Minus: V = A–V;
        break;
      case Times: V = A*V;
        break;
      default:
    }
    A = 0;
    F = Nop;
  }

  void pm() {
    V = –V;
  }
}
```

(b) On écrit:

```
public class State {
 ...

 void calc() {
   Touch current = new Touch();
   while (true) {
     current.touch();
     if (current.x != null) {
       A = V;
       V = current.x.val;
     }
     switch current.i {
     case mr:
       mr();
       break;
     case mp:
       mp();
       break;
     case clr:
```

```
        clr();
        break;
      case plus:
        plus();
        break;
      case moins:
        minus();
        break;
      case mult:
        times();
        break;
      case egal:
        equal();
        break;
      case pm:
        pm();
        break;
      default :
      }
    }
  }
}
```

(c) On utilise le mécanisme des exceptions. On doit d'abord créer un type d'exception correspondant. On peut par exemple (non requis pour la notation), se souvenir de l'opération que l'on voulait effectuer au moment où l'on s'aperçoit que l'accumulateur est vide:

```
public class AccVide extends Exception {
  Flag courant;
  AccVide(Flag f) {
    courant = f;
  }
}
```

Pour être complètement propre (non requis pour la notation non plus, on pouvait se contenter d'identifier un accumulateur à 0 avec un accumulateur vide), on peut changer `State` en:

```
public class State {
 Ent A;
 int V;
 int M;
 Flag F;
}
```

afin de repérer le fait que l'accumulateur est vide par le test `A==null`.

On doit maintenant changer les signatures et le code des fonction `plus`, `minus` et `times`, avec, par exemple:

```
 void plus() throws AccVide {
   equal();
   if (A==null) throw new AccVide(Plus);
   F = Plus;
 }
```

Puis il faut encapsuler le `switch/case` de la méthode `calc` dans un bloc:

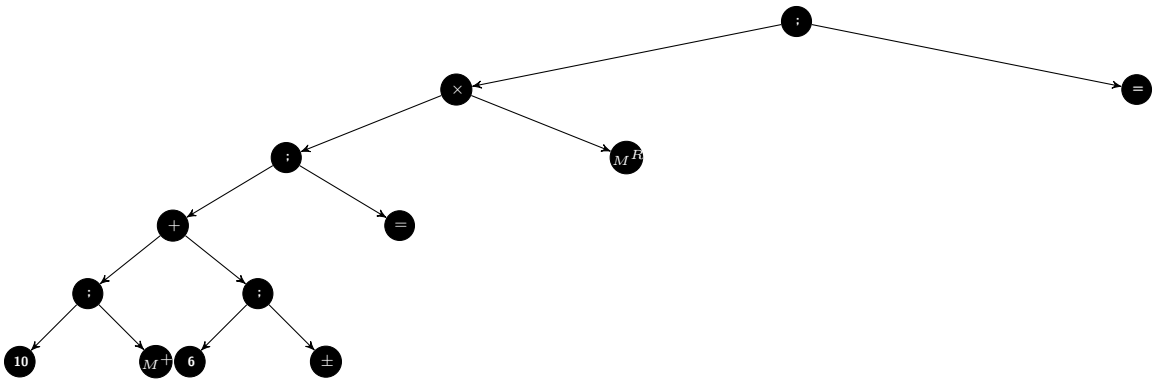

Figure 2: Arbre syntaxique correspondant au programme C

```
...
 try {
   ...
 } catch (AccVide e) {
    System.out.println("Accumulateur vide pendant l'op\'eration "
                                                    +e.courant+"!");
}
```

3 Calculatrice programmable

Devant la concurrence, le constructeur décide de rendre sa calculatrice "programmable": toute combinaison de touches peut être sauvegardée et rejouée plus tard. La grammaire d'un programme est:

$$\begin{aligned} program \quad ::= \quad & \emptyset \mid n \mid \pm \mid M^R \mid M^+ \mid program + program \mid program - program \\ & \mid program \times program \mid CLR \mid = \mid program; program \end{aligned}$$

donc est soit un programme vide ($\emptyset$), soit une constante, soit $\pm$, soit M^R, soit M^+, soit l'addition de deux programmes (c'est donc une définition récursive), soit la soustraction appliquée à deux programmes etc.

Pour stocker une combinaison de touches, l'utilisateur appuie sur *prgm* une première fois, passant en mode "programme", puis tape sa combinaison de touches. Il termine l'enregistrement de son programme en tapant à nouveau *prgm*. Il peut rejouer son programme courant en tapant *run*.

(5) (4 points) On souhaite dans cette question définir une classe JAVA, que l'on nommera `Program`, pour représenter un programme. La représentation choisie est celle d'un arbre de syntaxe. Par exemple, la combinaison de touches C, si elle est enregistrée comme programme, formera l'arbre de la figure 2.

(a) (1 point) Définir une classe Java *instruction élémentaire* `Instr`, pouvant prendre les valeurs M^+, M^R, $=$, $\pm$, CLR

(b) (1 point) Définir une classe `Node` pour représenter les noeuds de l'arbre de syntaxe. Il doit pouvoir contenir des valeurs représentant les trois opérations $+$, $-$, $\times$, mais aussi un séparateur entre plusieurs combinaisons de touches, que l'on écrira ;. Il devra pouvoir également indiquer, quand on est à une feuille de l'arbre, si on est une instruction élémentaire ou une constante entière (cf. figure 2)

(c) (1 point) Finalement, écrire la classe `Program`

(d) (1 point) Ecrire des constructeurs:

```
Program(Node n, Program exprl, Program exprr);
Program(int val);
Program(Instr i);
```

Le premier constructeur construit l'arbre correspondant à l'opération n (égale à $+$, $-$, ; ou $\times$) appliqué à une expression gauche `exprl` et à une expression droite `exprr`. Le deuxième construit l'arbre réduit à une feuille représentant une valeur numérique `val`. Enfin le dernier constructeur représente l'arbre réduit à une feuille représentant une instruction `Instr i`.

(6) (1 point) Ecrire une méthode *dynamique* `show` de la classe `Program` permettant d'afficher un programme sous la forme dont il a été rentré au clavier. Ainsi l'arbre de la figure 2 sera affiché sous la forme de la combinaison de touches C

$$10\ M^+\ +\ 6\ \pm\ =\ \times\ M^R =$$

Correction:

(5) (a) On définit:

```
public enum Instr { Mp, Mr, Eq, Pm, Clr };
```

(b) On définit:

```
public enum Node { Plus, Minus, Times, Then, Comm, Cste };
```

(c) Un `Program` est maintenant un arbre:

```
public class Program {
  Node n;
  Program exprl;
  Program exprr;
  int val;
  Instr i;
}
```

(d) Les constructeurs s'en suivent, simplement:

```
Program(Node ne, program exprle, program exprre) {
  n = ne;
  exprl = exprle;
  exprr = exprre;
}

Program(int vale) {
  n = Cste;
  val = vale;
}

Program(Instr ie) {
  n = Comm;
  i = ie;
}
```

(6) La méthode `show()` réalise un parcours infixe de l'arbre `Program`:

```
public class Program {
...
 void show() {
   switch n {
     case Comm:
       switch i {
         case Mp:
           System.out.print("M+ ");
           break;
         case Mr:
           System.out.print("MR ");
           break;
         case Eq:
           System.out.print("= ");
           break;
         case Pm:
           System.out.print("+/- ");
           break;
         case Clr:
           System.out.print("CLR ");
           break;
       }
       break;
     case Cste:
       System.out.print(val+" ");
       break;
     case Then:
       exprl.show();
       exprr.show();
       break;
     case Plus:
       exprl.show();
       System.out.print("+ ");
       exprr.show();
       break;
     case Minus:
       exprl.show();
       System.out.print("- ");
       exprr.show();
       break;
     case Times:
       exprl.show();
       System.out.print("* ");
       exprr.show();
       break;
     }
 }
}
```

4 Interpréteur de la calculatrice programmable

Pour donner une partie de la sémantique de la calculatrice programmable, on doit changer un peu la représentation de l'état courant de la calculatrice: c'est maintenant un *sextuplet* (A, V, M, F, P, E)

où (A, V, M, F) est comme avant un quadruplet contenant la valeur de l'accumulateur A, la valeur à l'affichage V, la valeur de la mémoire M, et l'opération courante dans $\{p, m, t, n\}$; E est un booléen indiquant si l'on est dans le *mode programme* (après avoir tapé une fois *prgm*, on en ressort en tapant de nouveau *prgm*) ou pas, et P est une expression dans la grammaire du début de la section 3, qui est le programme enregistré en mode programme.

On n'essaiera pas dans la suite de construire inductivement l'arbre instance de la classe Program de la section 3, correspondant au programme courant, qui nécessiterait de définir un état plus compliqué.

(7) ($\frac{1}{2}$ point) Que devrait renvoyer la suite de touches suivantes?

$$CLR\ 10\ +\ 6\ prgm\ +\ 4\ *\ 2\ prgm\ run$$

(8) (2 point) Ecrire la sémantique dénotationnelle d'un programme p donné. On exprimera donc inductivement sur la syntaxe du début de la section 3, la sémantique du programme.

(9) ($\frac{1}{2}$ point) Ecrire la sémantique des touches *prgm* et *run*. On supposera que taper *run* en cours d'écriture d'un programme revient à faire la suite de touches *prgm* puis *run*.

(10) (2 points) On souhaite dans cette question se donner les moyens de réécrire l'interpréteur de la calculatrice, sans avoir à reprogrammer ce que l'on avait déjà programmé à la question (4b)

(a) (1 point) Afin de minimiser le coût de développement, le constructeur veut réutiliser au possible son implémentation précédente. Par quel mécanisme peut-il espérer réutiliser le code de la question (4b) dans l'implémentation de la calculatrice programmable? Définir une classe codant l'état de la calculatrice programmable `Statep` à partir de la classe `State`

(b) (1 point) On écrira successivement les méthodes *dynamiques* de `Statep`, qui modifient *en place* l'état courant:

```
void prgm();
void run();
```

et qui interprètent respectivement l'action des touches *prgm* et *run*

Correction:

(7) 40.

(8) En appelant $[\![t]\!]$ la sémantique des touches de la question (3), agissant sur le quadruplet (A, V, M, F, P, E), on peut définir la sémantique $[\![.]\!]_p$ d'un programme inductivement sur sa syntaxe de la façon suivante:

$$\begin{aligned}
[\![\emptyset]\!]_p(A,V,M,F,P,E) &= (A,V,M,F,P,E)\\
[\![n]\!]_p(A,V,M,F,P,E) &= ([\![n]\!](A,V,M,F),P,E)\\
[\![\pm]\!]_p(A,V,M,F,P,E) &= ([\![\pm]\!](A,V,M,F),P,E)\\
[\![M^R]\!]_p(A,V,M,F,P,E) &= \left([\![M^R]\!](A,V,M,F),P,E\right)\\
[\![M^+]\!]_p(A,V,M,F,P,E) &= ([\![M^+]\!](A,V,M,F),P,E)\\
[\![clr]\!]_p(A,V,M,F,P,E) &= ([\![clr]\!](A,V,M,F),\emptyset,false)\\
[\![=]\!]_p(A,V,M,F,P,E) &= ([\![=]\!](A,V,M,F),P,E)\\
[\![p;q]\!]_p(A,V,M,F,P,E) &= [\![q]\!]_p\left([\![p]\!]_p(A,V,M,F,P,E)\right)\\
[\![p+q]\!]_p(A,V,M,F,P,E) &= [\![q]\!]_p\left([\![+]\!]\left(\pi_{14}([\![p]\!]_p(A,V,M,F,P,E))\right),\right.\\
&\qquad \left.\pi_{56}([\![p]\!]_p(A,V,M,F,P,E))\right)\\
[\![p-q]\!]_p(A,V,M,F,P,E) &= [\![q]\!]_p\left([\![-]\!]\left(\pi_{14}([\![p]\!]_p(A,V,M,F,P,E))\right),\right.\\
&\qquad \left.\pi_{56}([\![p]\!]_p(A,V,M,F,P,E))\right)\\
[\![p\times q]\!]_p(A,V,M,F,P,E) &= [\![q]\!]_p\left([\![\times]\!]\left(\pi_{14}([\![p]\!]_p(A,V,M,F,P,E))\right),\right.\\
&\qquad \left.\pi_{56}([\![p]\!]_p(A,V,M,F,P,E))\right)
\end{aligned}$$

où $\pi_{14}(A,V,M,F,P,E) = (A,V,M,F)$ et $\pi_{56}(A,V,M,F,P,E) = (P,E)$.

(9) On écrit:

$$\begin{aligned} [\![prgm]\!](A,V,M,F,P,E) &= (A,V,M,F,P,\neg E) \\ [\![run]\!](A,V,M,F,P,E) &= [\![P]\!](A,V,M,F,P,E) \end{aligned}$$

(on pourrait raffiner et interdire *run* quand on est en mode *prgm*).

(10) (a) On pourrait utiliser l'héritage. Par exemple, on écrirait:

```
public class Statep extends State {
  Program P;
  boolean E;
}
```

(b) Il suffit essentiellement de reprendre le code de `show`, et d'utiliser l'héritage et l'implémentation existante de $[\![t]\!]$ pour chaque action:

```
public class Statep extends State {
...

 void prgm() {
   E = !E;
 }

 void run() {
   if (P==null) return;
   switch P.n {
     case Comm:
       switch P.i {
         case Mp:
           mp();
           break;
         case Mr:
           mr();
           break;
         case Eq:
           equal();
           break;
         case Pm:
           pm();
           break;
         case Clr:
           clr();
           break;
       }
       break;
     case Cste:
       A = V;
       V = P.val;
       break;
     case Then:
       Program ps;
       ps = P;
       P = P.exprl;
       run();
```

```
            P = P.exprr;
            run();
            P = ps;
            break;
          case Plus:
            Program ps;
            ps = P;
            P = P.exprl;
            run();
            plus();
            P = P.exprr;
            run();
            P = ps;
            break;
          case Minus:
            Program ps;
            ps = P;
            P = P.exprl;
            run();
            minus();
            P = P.exprr;
            run();
            P = ps;
            break;
          case Times:
            Program ps;
            ps = P;
            P = P.exprl;
            run();
            times();
            P = P.exprr;
            run();
            P = ps;
            break;
          }
        }
      }
```

5 Calculatrice programmable élaborée (optionnel)

Le constructeur veut maintenant implémenter un mécanisme de boucle dans les programmes de la calculatrice de la partie 3. Il rajoute une touche *repeat* et *untilz*. La sémantique informelle en est que l'expression c dans *repeat* c *untilz* d sera répété tant que le nombre calcule par l'expression d est positif (strictement). On supposera que la condition d sera évaluée dans une mémoire supplémentaire, et donc n'écrase en aucun cas le calcul en cours, fait par le corps de la boucle c.

(11) (2 points) Considérant que la sémantique informelle nous indique que le programme suivant, que l'on nommera r:

repeat c *untilz* d

est équivalent (c étant un programme, corps de la boucle considérée, d étant un autre programme) à :

exécuter c
puis exécuter r si $d > 0$

donner la sémantique de r comme un problème de point fixe d'une fonctionnelle, dépendant de $[\![c]\!]$, sur un CPO que l'on définira brièvement. On n'essaiera pas de prouver de propriété particulière de cette fonctionnelle (comme la croissance ou la continuité)

(12) (2 points) Le service marketing essaie d'élaborer un argumentaire de vente. Quelles fonctions supplémentaires peut-on calculer avec ce mécanisme? Se contenter de donner un exemple de fonction nouvelle (par rapport aux expressions polynômiales simples accessibles jusqu'alors) et d'une fonction *a priori* pas programmable même avec ce nouveau mécanisme (sans preuve)

Correction:

(11) L'état de la calculatrice comprend une mémoire supplémentaire R, S, T (l'équivalent de A, de V, et de F), on considère maintenant des nuplets $(A, V, M, F, P, E, R, S, T)$. On écrit l'équivalence définissant le *repeat until* comme une équation de plus petit point fixe. Soit $r = repeat\ c\ untilz\ d$, alors:

$$[\![r]\!]_q(A,V,M,F,P,E,R,S,T) \quad =$$

$$\begin{cases} [\![r]\!]_q\left([\![c]\!]_q(A,V,M,F,P,E,R,S,T)\right) & \text{si } [\![d]\!]\left([\![c]\!](A,V,M,F,P,E,R,S,T)\right) > 0 \\ & \text{sinon} \end{cases}$$

Avec,

$$[\![d]\!]_q(A,V,M,F,P,E,R,S,T) \quad = \quad (A,V,M,F,P,E,[\![d]\!](R,S,M,T))$$

où $[\![.]\!]$ est la sémantique de la question (3).

(12) On peut par exemple maintenant calculer une factorielle:

repeat M^R $\times$ untilz $M^R - 1$ M^+

Une fonction a priori non programmable même avec ce nouveau mécanisme est une fonction non calculable au sens classique du terme, mais aussi toute fonction récursive partielle non récursive primitive (fonction d'Ackerman par exemple).