***Corrigé / Examen TPGO 2019-2020***

**Exercice 1**

Chaque $C_i$ peut être remplacé par une conjonction de clauses à au plus 3 littéraux de la manière suivante :
Cas k =1 : Ajouter à chaque $C_i$ avec une variable ($C_i$= x) deux variables $a_i$ et $b_i$
Remplacer x
Par $(x \vee a_i \vee b_i)\ (x \vee a_i \vee b'_i)\ (x \vee a'_i \vee b_i)\ (x \vee a'_i \vee b'_i)$

Cas k=2 : Ajouter à chaque $C_i$ à deux variables ($C_i$= x V y) une variable $c_i$
Remplacer x V y
Par $(x \vee y \vee c_i)\ (x \vee y \vee c'_i)$

Ces deux premiers cas sont simples à montrer.
Par exemple, pour k=2 il suffit de poser
$E=(x \vee y \vee c_i)\ (x \vee y \vee c'_i)$
Cette dernière expression est équivalente à
$E= (x+y+c_i)(x+y+c'_i)$
Posons z=x+y
$E= (z+c_i)(z+c'_i)$
$E= zz + zc_i+zc'_i+c_ic'_i$
$E= 1 + z(c_i+c'_i) + 0$
$E= 1 + z\ (1) + 0$
$E= z$
$E= x + y$

Cas k>3Chaque clause $C = x_1 \vee x_2 \vee \ldots \vee x_k$ devient
$C_0 = (x_1 \vee x_2 \vee y_1)\ (y'_1 \vee x_3 \vee y_2)\ (y'_2 \vee x_4 \vee y_3)\ldots(y'_{k-3} \vee x_{k-1} \vee x_k)$

Montrons les deux implications.
a) $C_0$ satisfiable → C satisfiable
Supposons que les clauses de $C_0$ sont toute satisfaites.
Donc, au moins un des littéraux $x_1, x_2, \ldots, x_k$ doit être vrai.
Autrement $y_1$ est vrai, ce qui force $y_2$ à vrai et ainsi de suite. La dernière clause serait fausse.
L'implication est donc montrée.
b) C satisfiable → $C_0$ satisfiable
$x_1 \vee x_2 \vee \ldots \vee x_k$ satisfaite veut dire qu'il existe au moins un i tel que $x_i$ = Vrai. On pose $y_1$, $y_2, \ldots, y_{i-2}$ tous à Vrai et le reste à Faux. Ceci garantit que les clauses de $C_0$ soient vraies.

**Exercice 2**

A) Définition d'un λ-terme:
- Une constante est un λ_terme
- Une variable est un λ-terme
- Si u et v sont des λ-termes alors uv est un λ-terme
- Si x est une variable et u un λ-terme alors λx.u est un λ-terme

B) Réduction
$AA = (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))\ (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))$
$AA = \lambda y.y\ (\quad (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))\quad (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))\ y\ )$
$BM = AAM = M\ (\quad (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))\quad (\lambda\ x\ y\ .\ y\ (\ x\ x\ y\ ))\ M\quad ) = M\ (AA)\ M = M\ (BM).$

C) Déduction
BM est solution de l'équation à point fixe X = M X
C'est ce qu'on appelle un combinateur du point fixe

**Exercice 3**

A) Vérification

Les algorithmes qui suivent vérifient si l'ensemble V est une couverture de sommets de taille k (pour un graphe G (S, A) avec S:ensemble des sommets et A:ensemble des arêtes):

Algorithme 1

```
Compte := 1
Pour chaque sommet v dans V
    Supprimer de l'ensemble A toutes les arêtes adjacentes à v
    Compte := Compte + 1
    Si compte = k et E est vide
        V est une couverture de taille k
    Sinon
        V n'est pas une couverture de taille k
```

Algorithme 2

```
Si |V| = k
    Pour chaque arête (u, v) de A
        Si u n'est pas dans V et v n'est pas dans V
            V n'est pas une couverture de taille k
Sinon
    V' n'est pas une couverture de taille k
```

Il est facile de voir que les deux algorithmes s'exécutent en temps polynomial (O(nm) avec n: nombre de nœuds dans V et m le nombre d'arêtes du graphe

B) Solution exacte : recherche d'une couverture de longueur minimale avec un parcours en largeur
Soit Noeuds [ 1..N] le tableau des nœuds.

```
Arrêt := faux
CREERFILE ( F )
ENFILER ( F , '' )
TQ NON FILEVIDE ( F ) ET NON Arrêt
    DEFILER ( F , Une_branche )
    Si Une_branche est une couverture
        Couverture cherchée := Une_branche
        Arrêt := Vrai
    SINON
        POUR i := Longueur(une_branche) + 1 , N
            ENFILER ( F , Une_branche + Noeuds [ i] ) )
        FPOUR
    FSI
FTQ
```

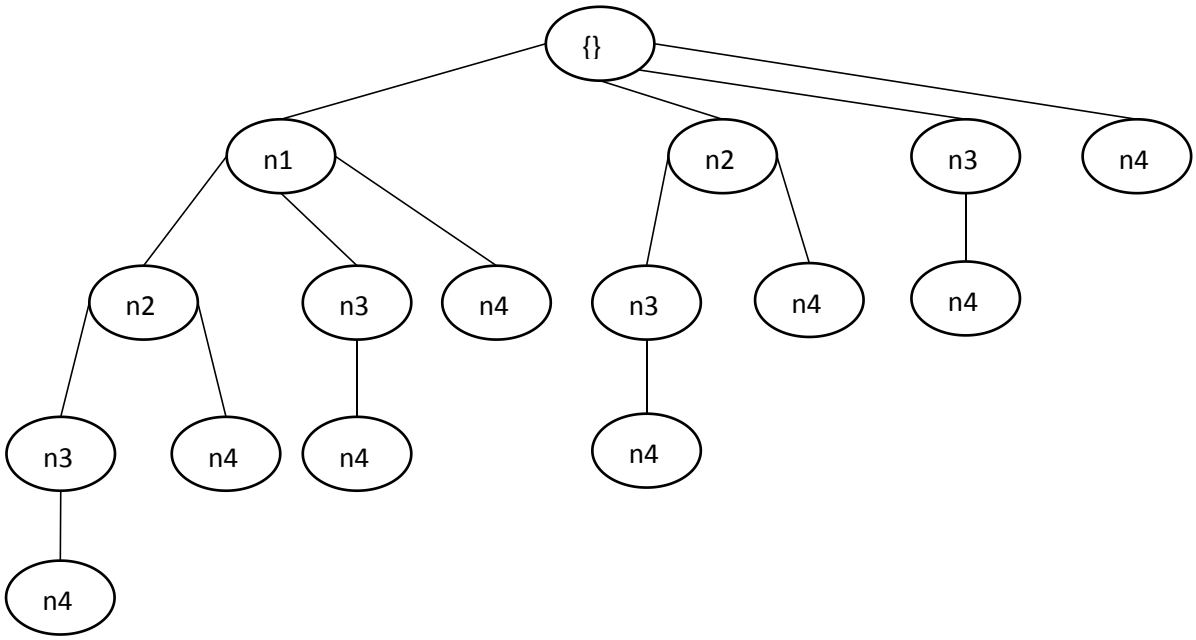
La file d'attente contient les branches de l'arbre. Les branches contiennent des sous ensembles.

A chaque fois qu'on défile une branche, on commence par vérifier si les nœuds la composant forment une couverture.

Dés que la première couverture est trouvée, l'algorithme s'arrête, puisque l'algorithme (parcours en largeur) traite dans l'ordre les sous ensembles de longueur 1, puis les sous ensembles de longueur 2, et ainsi de suite.

Dans le cas où la branche défilée n'est pas une couverture, on crée de nouvelles branches en étendant la branche qui vient d'être défilée par les nœuds de rang supérieur et les enfiler. De cette manière, il n y a pas formation de cycles dans les branches. Par contre on peut re visiter des nœuds déjà visités.

Si l'ensemble des sommets est {n1, n2, n3, n4}, l'arbre généré par l'algorithme est le suivant:




Les branches de niveau 1 sont : n1, n2, n3 et n4

Les branches du niveau 2 sont n1n2, n1n3, n1n4, n2n3, n2n4 et n3n4

Les branches du niveau 3sont n1n2n3, n1n2n4, n1n3n4 et n2n3n4

La branche du niveau 4 est n1n2n3n4.

Énumération de tous les cas possibles

Dans le pire des cas, la couverture est de longueur la cardinalité de l'ensemble des sommets du graphe. Et dans ce cas:

Considérer tous les sous-ensembles formés d'un élément et vérifier s'il existe un ensemble couvrant. Il y a $C_n^1 = n$

Considérer tous les sous-ensembles formés de deux éléments et vérifier s'il existe un ensemble couvrant. Il existe $C_n^2$

Considérer tous les sous-ensembles formés de trois éléments et vérifier s'il existe un ensemble couvrant. Il existe $C_n^3$

Etc.

Considérer aussi le cas de l'ensemble plein ($C_n^n=1$)

Le nombre de sous ensembles à vérifier est donc $C_n^1 + C_n^2 + .....C_n^n = 2^n - 1$

C) Solution approximative : recherche d'une couverture avec un parcours Best First Search
Soit Noeuds [ 1..N] le tableau des nœuds et Degres[1..N] le tableau des degrés correspondants

```
CREERFILE ( FP ) ;
ENFILER ( FP , '' , 0) ;
Arrêt := Faux
TQ NON FILEVIDE ( FP ) et NON Arrêt
    DEFILER ( FP , Une_branche ) ;
    SI Une_branche est une couverture
        Couverture cherchée := Une_branche
        Arrêt := Vrai
    SINON
        POUR i := Longueur(une_branche) + 1 , N
            ENFILER ( FP , Une_branche + Noeuds [ i] ), Degre[i] )
        FPOUR
    FSI
FTQ
```

La file d'attente avec priorité FP contient les branches de l'arbre ordonnées en ordre décroissant selon le degré du dernier nœud de la branche. Les branches contiennent des sous ensembles.

A chaque fois qu'on défile une branche, on crée de nouvelles branches en étendant la branche supprimée par les nœuds de rang supérieur. On les enfile selon le degré des nœuds rajoutés. De même, il n y a pas formation de cycles dans les branches et les nœuds déjà visités peuvent être re visités.

D) Classe du problème de la couverture des sommets

On ne connaît pas un algorithme polynomial pour le problème donné. L'algorithme exacte connu est exponentiel ($O(2^n)$. Par contre, il existe un vérificateur, qui s'exécute en temps polynomial, capable de répondre par oui ou non à une instance donnée du problème. Le problème de la couverture des sommets est donc dans la classe NP.

E) Ci-après, quelques solutions gloutonnes ( heuristiques) permettant de trouver une couverture ( pas nécessairement la plus petite)

Heuristique 1

```
C = {}
Considérons l'ensemble A de toutes les arêtes du graphe donné.
Tant que A n'est pas vide
    a) Choisir une arête arbitraire (u, v) de l'ensemble A et ajoutez 'u' et 'v' à C.
    b) Retirez toutes les arêtes de A qui sont incidentes sur u ou v.
Retourner C
```

Heuristique 2

```
C := {}
Pour chaque nœud u dans S
  Si u n'appartient pas à C et u possède un nœud adjacent v n'appartenant pas à C
     Alors rajouter u et v dans C
Retourner C
```

Heuristique 3

```
C := {}
Pour chaque nœud u dans S
    Si u possède au moins un nœud adjacent v tel que Rang(v) > Rang(u)
        Alors rajouter u dans C
Retourner C
```

Heuristique 4

```
C := {}
```

*E' := E*
*Tq E # {}*
*Prendre le nœud v avec le plus grand degré*
*C := C + {v}*
*Éliminer v et toutes les arêtes incidentes à v*
*Ftq*
*Retourner C*

Heuristique 5
*Mettre les nœuds dans un tableau V[1..N] (ou liste)*
*Trier V selon le degré en ordre croissant.*
*Pour chaque nœud u € V [1..N]*
*Supprimer u de V [1..N] si tous ses nœuds adjacents existent dans V [1..N].*
*Si u est supprimé, faire N=N-1*
*Retourner V [1..N]*

**Exercice 4**

A) Algorithme

```
x:=1;
n:=N;
Tq n > 0 :
    x:=x*n;
    n := n-1
Ftq
Retourner (x)
```

B) Preuve

Il s'agit de prouver

$N \geq 0$ { x:=1; n:=N; Tq n>0 : x:=x*n; n := n-1Ftq} (x=N!)

Ou encore

Vrai { x:=1; n:=N; Tq n>0 : x:=x*n; n := n-1Ftq} (x=N!) (1)

Pour prouver (1), il faut prouver par (SEQ)

Vrai {x:=1; n := N} E (2) et

E {Tq n>0 : x:=x*n; n := n-1Ftq} (x=N!) (3)

Pour prouver (3) il faut prouver par (IMP)

E {Tq n>0 : x:=x*n; n := n-1Ftq} E & ($n \leq 0$) (4) et

E & ($n \leq 0$) → (x=N!) (5)

Pour prouver (4), il faut prouver par (ITE)

E & (n>0) {x:=x*n; n:=n-1} E (6)

Pour prouver (6), il faut prouver par (SEQ)

E & (n>0) { x:= x*n } F (7)

F {n:=n-1} E (8)

Pour prouver (7) il faut prouver par (IMP)

E & (n>0) → G (9) et

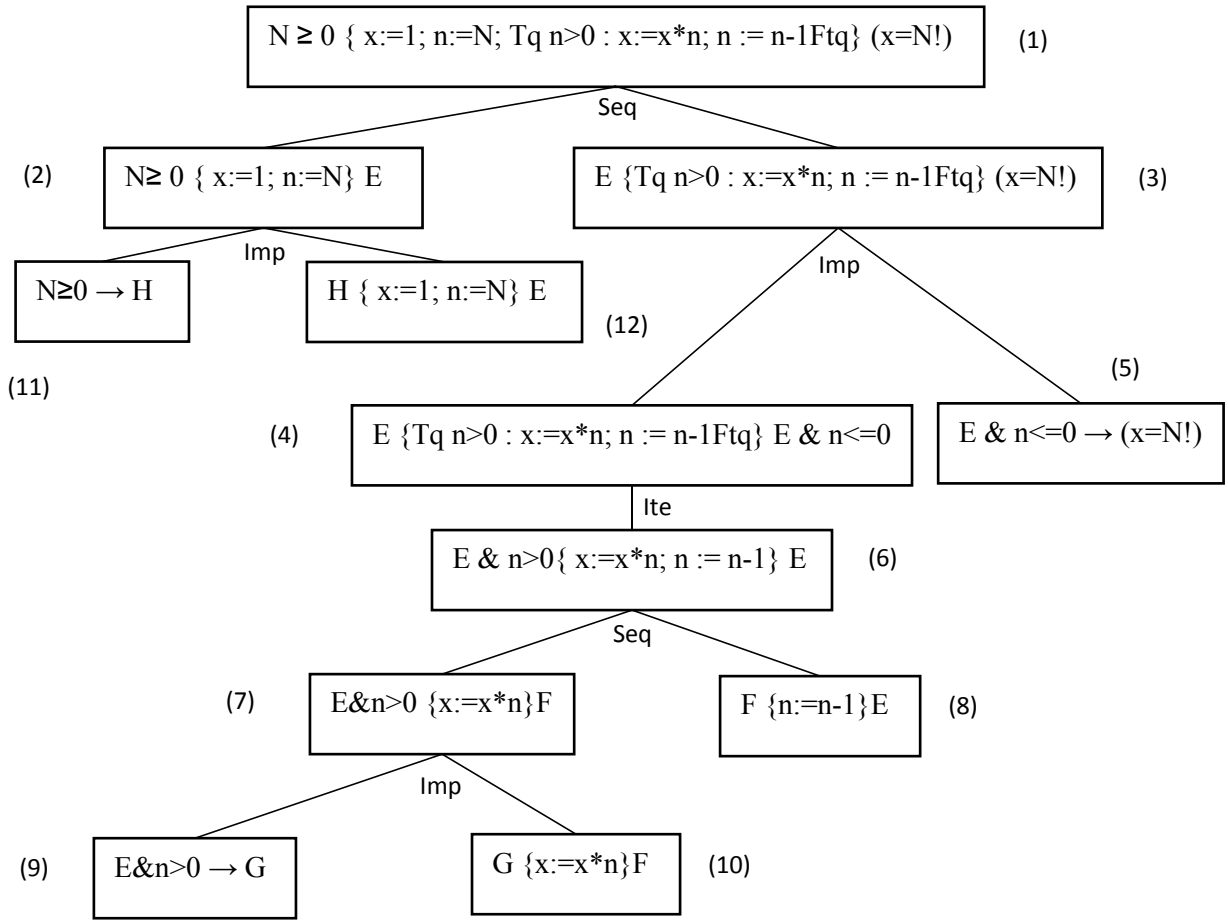
G{x:=x*n} F (10)

Enfin, pour montrer (2), il faut prouver par (IMP)

$N \geq 0$ →H (11) et

H { x:=1; n:=N} E (12)

L'arbre de preuve est donc:




A ce niveau, nous proposons comme invariant E ≡ ( x*n!=N!)
Il faut montrer que
- E est un invariant de boucle
- l'implication (5) est vraie
Ensuite, déterminer F et G pour montrer que
- l'implication (9) est vraie
Enfin montrer que (2) est vraie.

<u>E est bien un invariant</u> car l'application de l'affectation n:=n-1 donne
(x*(n-1)!=N!) {n:=n-1} (x*n!=N!)
Et l'application de l'affectation x:=x.n donne
(x*n*(n-1)!=N!){ x:=x*n} (x*(n-1)!=N!)
C'est à dire
(x*n!=N!){ x:=x*n} (x*(n-1)!=N!)

<u>Implication(5)</u>
Pour n≤0, on a n! = 1 et donc( x*n! = N!) ≡ (x=N!)

<u>Détermination de F et G</u>
F ≡ (x*n*(n-1)!=N!)
G ≡ (x*n!=N!)

Implication (9)
Pour (n>0) on a bien ( x*n!=N!) ≡ (x*n!=N!)

Montrer (2)
Par (AFF) on a
( x*N!=N!){ n := N} ( x*n!=N!)
Par (AFF) on a
( 1*N!=N!) { x:=1} ( x*N!=N!)
Qui est Vrai.

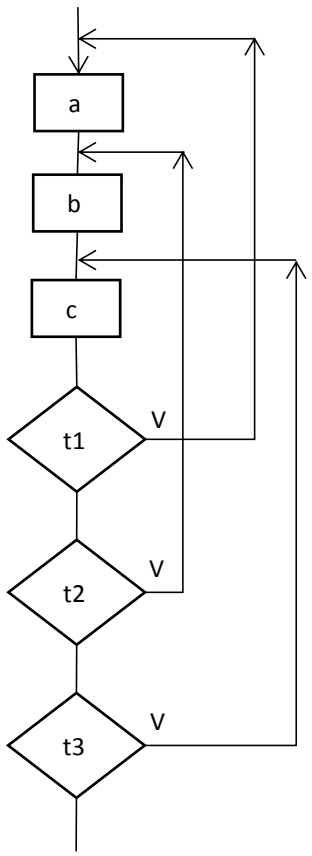
**Exercice 5**

A) Point fixe :

Et1 : a ;
Et2 : b ;
Et3 : c ;
if t1 goto Et1 ;
if t2 goto Et2 ;
if t3 goto Et3 ;
Stop

B) L'organigramme correspondant $O$ est le suivant:

$O$

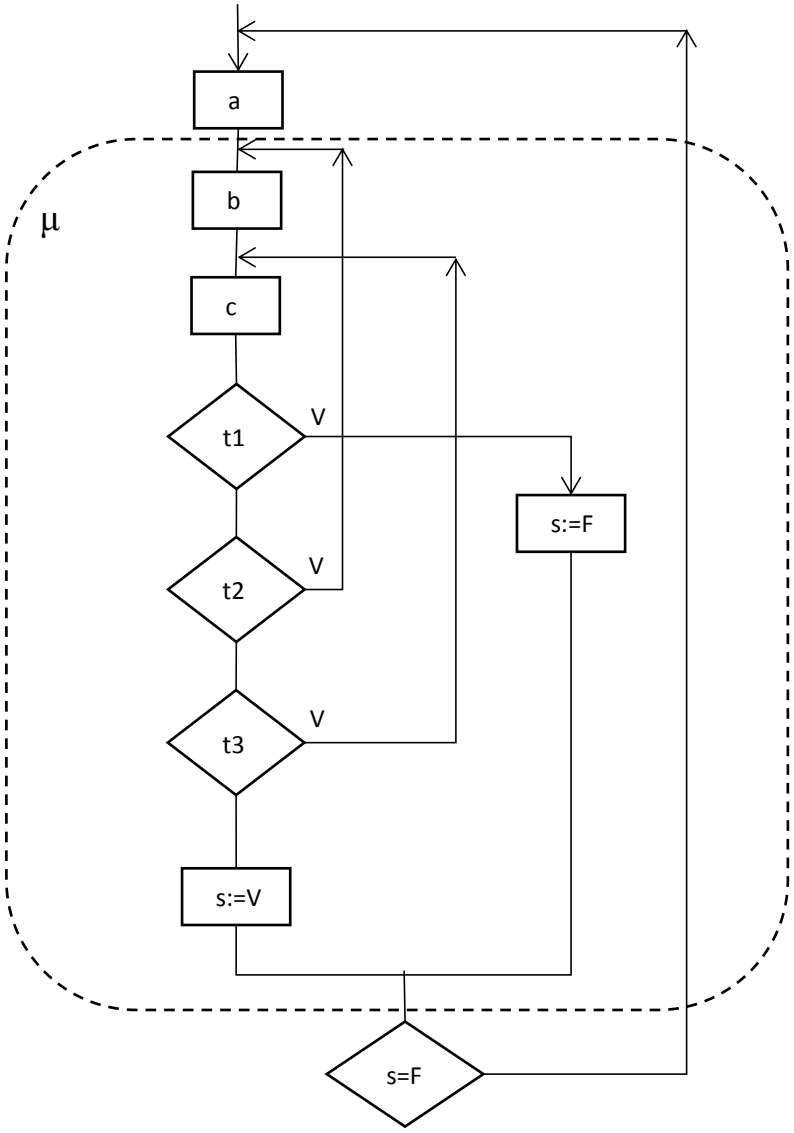


C) Transformation

L'organigramme $O$ est du type 1. Il y a un arc qui remonte vers l'action a et un arc qui sort du test t3 pour la valeur fausse de t3. Dans les organigrammes, une rectangle désigne une action ou un groupe d'actions et un losange désigne un test. V désigne Vrai et F désigne Faux.

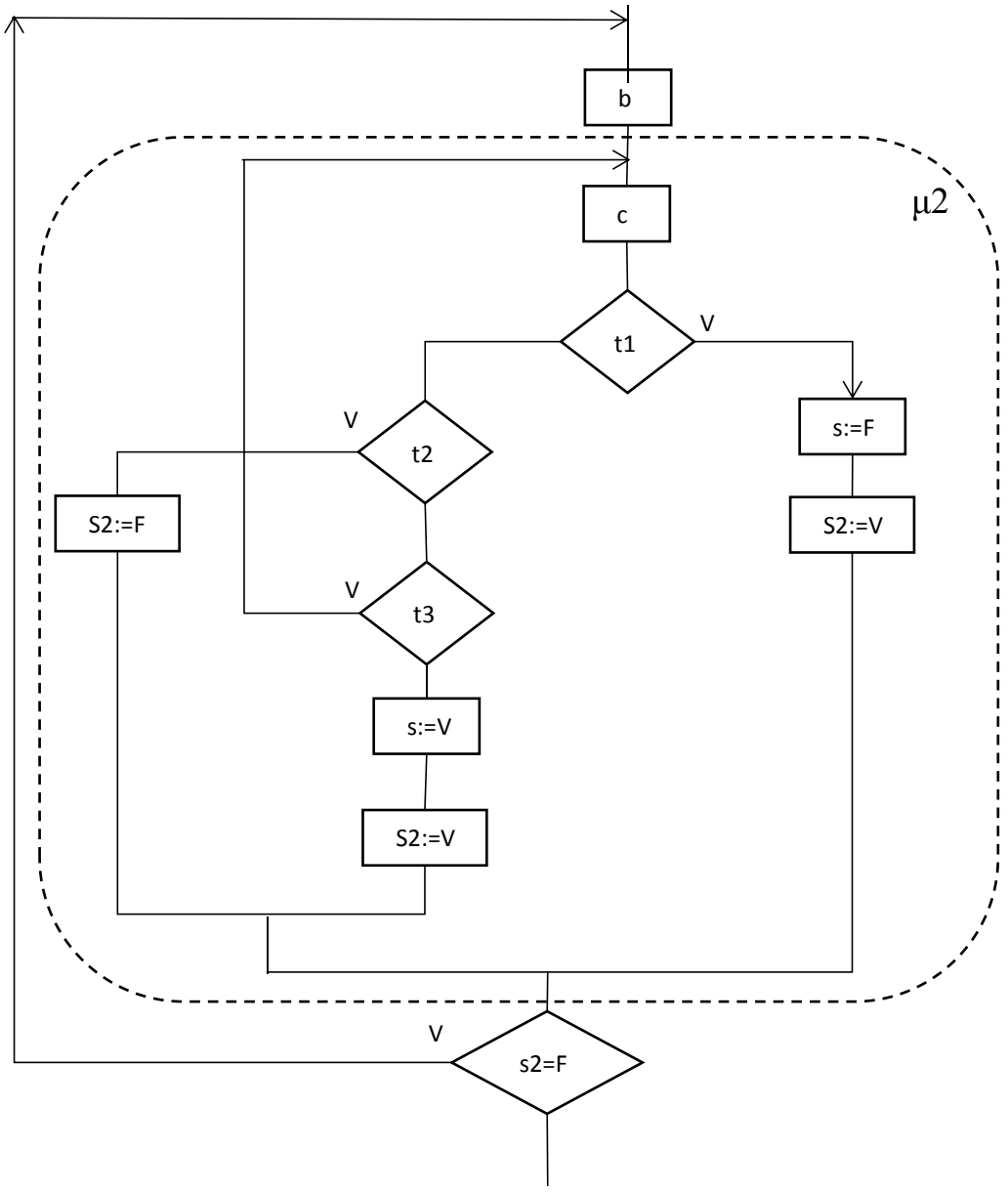
Il est transformé selon l'organigramme ci-après et a pour code
*O*=

Repeat
a
μ
Until s



L'organigramme μ est aussi du type 1. Il y a un arc qui remonte vers l'action b et deux arcs qui sortent ( après s:=V et s:= F).
Il est transformé selon l'organigramme ci-après et a pour code

μ=
Repeat
b
μ2
Until s2




L'organigramme μ2 est aussi du type 1. Il y a un arc qui remonte vers l'action c et trois arcs qui sortent ( après s2:=V, s2=V et s2:= F).

Il est transformé selon l'organigramme ci-après et a pour code

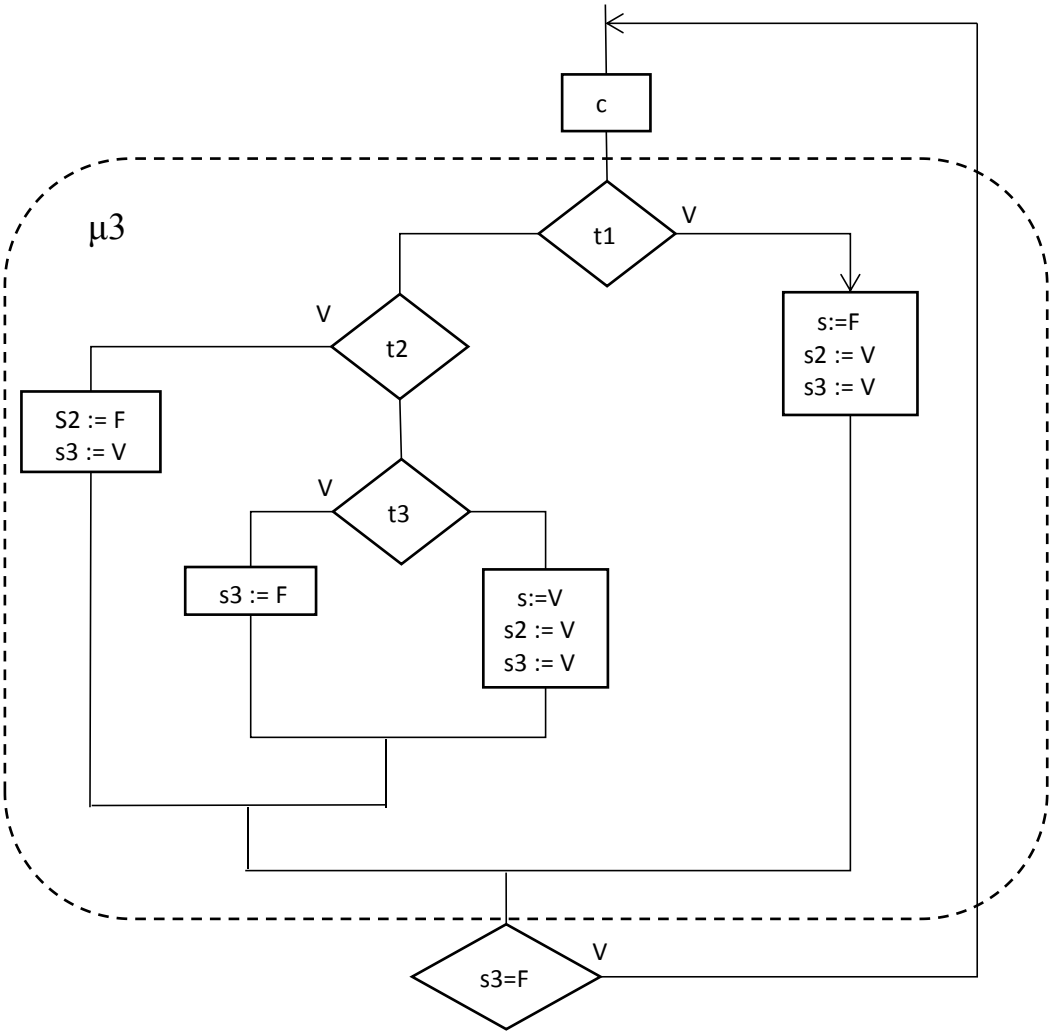
μ2 =

Repeat

c

μ3

Until s3




Le code correspondant à μ3 est structuré et est le suivant:

```
Si t1
    s:=F; s2:=V; s3:= V
Sinon
    Si t2
        s2 := F; s3:=V
    Sinon
        Si t3
            s3 := F
        Sinon
            s:=V; s2:=V; s3:= V
        Fsi
    Fsi
Fsi
```

Le code final est:

*O*=

```
Repeat
    a
    Repeat
        b
        Repeat
            c
            μ3
        Until s3
    Until s2
Until s
```

D) Schéma associé : 3 boucles emboîtées.