Chapitre 2 / 2e Partie / TPGO

Programmation Procédurale

A. Introduction

B. Schémas et transformations

C. Preuves formelles

Objectif : Etude du paradigme 'Procédural'

A. Introduction

Programmation Procédurale / Impérative / Conventionnelle

Décrire la procédure à suivre pour résoudre un problème

Un style (paradigme) de programmation de bas niveau

Généralement proche de l'architecture de la machine

Basé sur les changements d'état de la mémoire

Caractéristiques

Affectations

Séquencement contrôlé

Vision opérationnelle des solutions indiquant pas à pas la procédure à suivre

A. Exemple

Solution procédurale pour le calcul du produit scalaire de 2 vecteurs

```
Soient A et B deux vecteurs de taille n
Soit Res une variable réelle
Soit i une variable entière
Debut
    Res ← 0 ;
    i ← 1 ;
    TantQue ( i <= n )
        Res ← Res + A[i] * B[i] ;
        i++
    FTQ
Fin
```

Remarques :

- Présence de **variables étrangères au problème**
- Introduction d'un **séquencement artificiel** dans le déroulement des calculs

A. Conséquences

Prise en compte dans les solutions de considérations étrangères au problème à résoudre, liées principalement à l'architecture de la machine utilisée

→ Beaucoup de variables
→ Beaucoup de changements d'états difficiles à suivre
→ Ordres (séquencement) de calcul imposés

Deux grandes conséquences :

1) Raisonnement sur programmes difficile

2) Mise en Œuvre généralement efficace

B. Schémas / Introduction

Le degré de liberté donné au programmeur quant au contrôle du séquencement de l'exécution permet de distinguer **2 classes** de langages procéduraux :

1- Les langages structurés
2- Les langages non structurés

Afin d'étudier l'influence de ce degré de liberté dans la programmation, des langages abstraits ont été définis et comparés

→ **Les schémas de programmes**

Ces schémas de programmes sont construits avec des symboles de constantes, de fonctions et de prédicats

Nous distinguons deux types d'instructions :

1- Les tests ou prédicats (P) et les actions simples (A) comme les affectations, lecture/écriture, ...)

2- Les structures de contrôles

B. Définition de Schémas

Un schéma de programme peut être représenté graphiquement par un organigramme non interprété, utilisant uniquement des symboles d'actions (A) et de tests (P) :

B-Schémas

Cette structure représente les langages à branchement non structurés.

Ils sont définis comme étant la plus petite solution de l'équation à point fixe suivante :

B = *A* ∪ *etiq:**B*** ∪ ***B;B*** ∪ *si P aller_à etiq* ∪ *aller_à etiq*

D-Schéma

Cette structure représente les langages itératifs structurés (schéma de Dijkstra)

Ils sont définis comme étant la plus petite solution de l'équation à point fixe suivante :

D = *A* ∪ ***D;D*** ∪ *si P alors **D** sinon **D** fsi* ∪ *tantque P faire **D** fait*

B. Définition de Schémas (suite)

BJ_n-Schémas

C'est une classe de structures représentant les langages itératifs semi-structurés (introduits par Böhm et Jacopini).

Ils sont définis comme étant la plus petite solution de l'équation à point fixe suivante :

BJ = A ∪ **BJ**;**BJ** ∪ si P alors **BJ** sinon **BJ** fsi ∪ faire P → **BJ** ; … P → **BJ** fait

Le corps de la boucle faire … fait contient un certain nombre (k) de blocs P → BJ, (avec $1 \leq k \leq n$).

Chacun d'eux est interprété comme suit :

Si le prédicat P est vrai
on exécute le blocs BJ associé et on passe au prochain P → BJ
ou alors on boucle si on est arrivé au dernier.
Sinon
on sort de la boucle faire … fait

Remarque : BJ_1 n'est rien d'autre qu'un D schéma

B. Définition de Schémas (suite)

RE_n-Schémas

C'est une classe de structures représentant les langages itératifs semi-structurés introduisant les branchements de sortie de boucle (instruction *exit( )*).

Ils sont définis comme étant la plus petite solution de l'équation à point fixe suivante :

RE = A ∪ **RE**;**RE** ∪ si P alors **RE** sinon **RE** fsi ∪ faire **RE** fait

Le corps de la boucle *faire … fait* peut contenir des instruction 'exit(i)' pour sortir de i boucles englobantes, avec $1 \leq i \leq n$.

REC_n-Schémas

C'est une classe de structures représentant les langages itératifs semi-structurés utilisant en même temps les branchements de d'entrée et de sortie de boucle.

Définies avec la même équation à point fixe que les RE_n-Schémas.

La différence avec les RE_n-Schémas réside dans la possibilité d'utiliser, en plus, dans le cors d'une boucle *faire … fait*, l'instruction '*entrer(i)*' pour aller au début de la $i^{ième}$ boucle englobante

B. Interprétation de schémas

Pour donner un sens (sémantique) à un schéma de programme, il est nécessaire d'attribuer une interprétation à chaque symbole utilisé. Cela consiste donc à choisir :

- un ensemble de discours donné D
- un élément de D pour chaque symbole de Constantes
- une fonction de $D^n \rightarrow D$ pour chaque symbole de fonction
- une fonction de $D^n \rightarrow \{ v, f \}$ pour chaque symbole de prédicat

Appliquer une interprétation I sur un schéma de programme π donne lieux à un programme

$(I[\![\pi]\!])$

Exemple de D-Schéma : soit π le schéma de programme suivant :

```
x ← a ; y ← b ; lire(z)          // a et b des symboles de constantes
TantQue ( p(x,z) )               // x, y et z des symboles de variables
     y ← f(x,y) ; x ← g(x)       // p un symbole de prédicat
FTQ                              // f et g des symboles de fonctions
```

Soit I1={ D → N, a → 1, b → 0, p(x,z) → x < z, f(x,y) → x+y, g(x) → x+1}

$I1[\![\pi]\!]$: représente le programme qui calcule la somme des z premiers entiers

Soit I2={ D → N, a → 1, b → 1, p(x,z) → x <= z, f(x,y) → 2*y, g(x) → x+1}

$I2[\![\pi]\!]$: représente le programme qui calcule 2^z

B. Réductions entre structures

On définit 2 relations R1 et R2 entre schémas de programmes :

a) $\pi 1$ R1 $\pi 2$: $\forall$ I une interprétation de $\pi 1$ et $\pi 2$, on a $I[\![\pi 1]\!] = I[\![\pi 2]\!]$.

b) $\pi 1$ R2 $\pi 2$: Les ensembles d'actions A et de prédicats P ayant une occurrence dans $\pi 1$ ou $\pi 2$, sont les mêmes ($\pi 1$ et $\pi 2$ sont construits avec les mêmes ensembles A et P).

Soient S1 et S2 deux structures et R une relation.

On dit que **S1 est réductible à S2** par la relation **R** (et on note **$S1 \leqslant_R S2$**)
si : $\forall \pi 1 \in S1, \exists \pi 2 \in S2$ /que $\pi 1$ R $\pi 2$

On dit que **S1 est équivalent à S2** par la relation **R** (et on note **$S1 =_R S2$**)
si : $S1 \leqslant_R S2$ et $S2 \leqslant_R S1$

On dit que **S1 est strictement réductible à S2** par la relation **R** (et on note **$S1 <_R S2$**) si : $S1 \leqslant_R S2$ et Non($S2 \leqslant_R S1$)

B. Comparaison entre structures

A partir de R1 et R2, on définit 2 types de comparaisons entre structures :

1) Conversion Fonctionnelle (FN)

C'est une réduction par la relation R1

S1 $\leqslant_{FN}$ S2 : $\forall$ I une interprétation, $\forall \pi 1 \in S1, \exists \pi 2 \in S2$ /que $I[\![\pi 1]\!] = I[\![\pi 2]\!]$.
Informellement, pour tout programme pouvant être écrit avec le langage S1, il existe toujours un programme équivalent en langage S2.

2) Conversion Sémantique (SE)

C'est une réduction par la relation (R1 & R2)

S1 $\leqslant_{SE}$ S2 : $\forall$ I une interprétation, $\forall \pi 1 \in S1, \exists \pi 2 \in S2$ ayant même ensembles A et P (que $\pi 1$) /que $I[\![\pi 1]\!] = I[\![\pi 2]\!]$.
Informellement, pour tout programme pouvant être écrit avec le langage S1, il existe toujours un programme équivalent en langage S2 en réutilisant les mêmes actions simples et tests.

B. Quelques résultats importants

- Comparaison sémantique (Rao-Kosaraju 1974)

$D =_{SE} BJ1 <_{SE} BJ2 <_{SE} \dots Bj_{\infty} <_{SE} RE_1 <_{SE} RE_2 \dots <_{SE} Re_{\infty}$

$=_{SE} \quad =_{SE} \quad =_{SE}$

$REC_1 <_{SE} REC_2 \dots <_{SE} REC_{\infty} =_{SE} B$

- Comparaison Fonctionnelle (Böhm et Jacopini 1968)

Théorème : $D =_{FN} B$

(voir démonstration plus loin)

B. Transformations

Une transformation prend en entrée un programme source et produit en sortie un programme source

- Utilisée dans les réductions entre structures

Les transformations peuvent être utilisées pour différentes raisons :

- Synthèse de programmes
- Structuration de programmes
- Etude / Preuve de propriétés

B. Quelques Transformations

1) Böhm et Jacopini 1968

B → D en utilisant l'identification de patterns dans les organigrammes

2) Arsac 1977

B → RE_{∞}

Par résolution d'un système d'équations où les variables identifient les blocs étiquetés

3) Williams et Chen 1985

Élimine les goto dans un programme Pascal (un genre simplifié de B → D)

La méthode repose sur l'identification de certains schéma contenant des goto (en Pascal) et leur équivalents sans instructions de branchement.

4) Passage par automate 1996 (à l'INI – projet CONCORD)

B → D

Les étiquettes sont vues comme des états et les branchements comme des transitions enrichis par les éventuelles conditions et actions à réaliser avant le branchement.

La matrice de transition permet alors de guider la transformation comme dans une analyse d'un mot en entrée d'un automate.

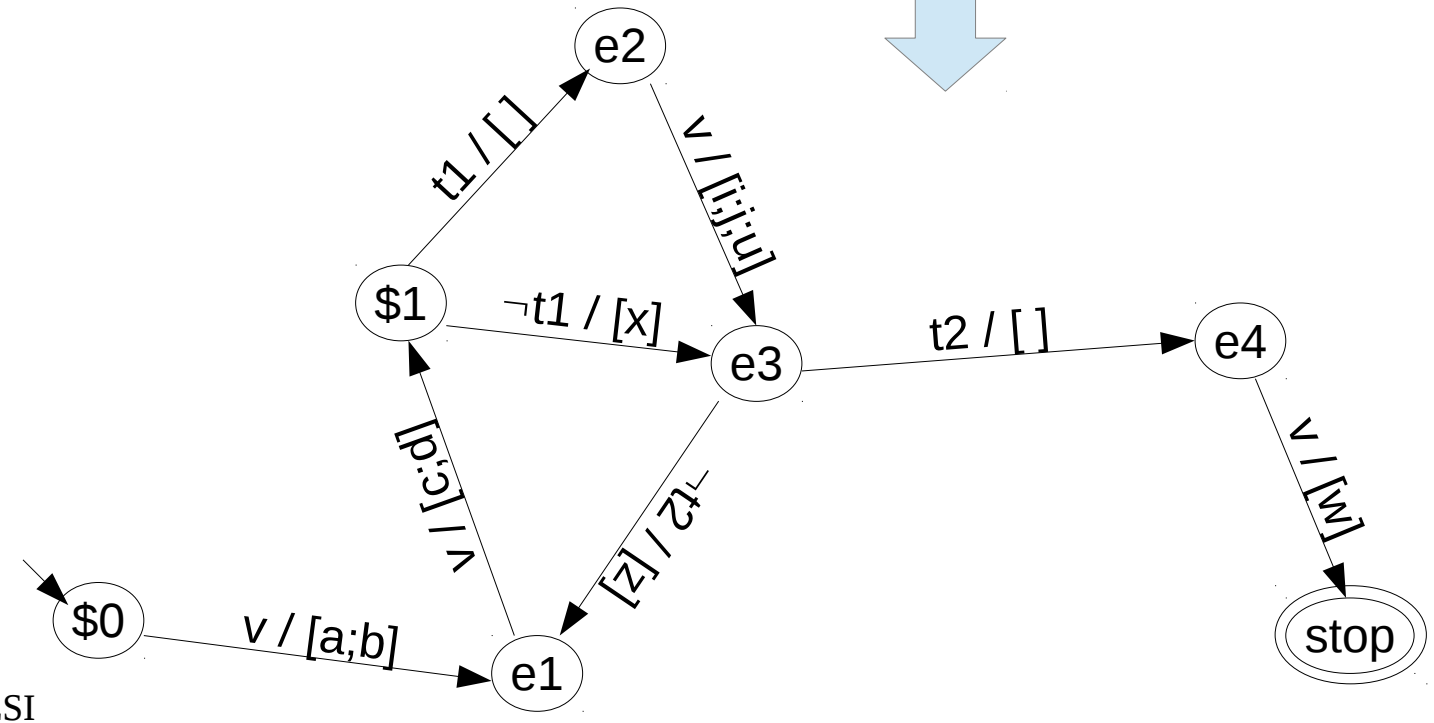
Exemple de transformation par automate : B → D

```
        a ;
        b ;
e1 :    c ;
        d ;
        if t1 goto e2 ;
        x ;
e3 :    if t2 goto e4 ;
        z ;
        goto e1 ;
e2 :    i ;
        j ;
        u ;
        goto e3 ;
e4 :    w;
        stop ;
```

1

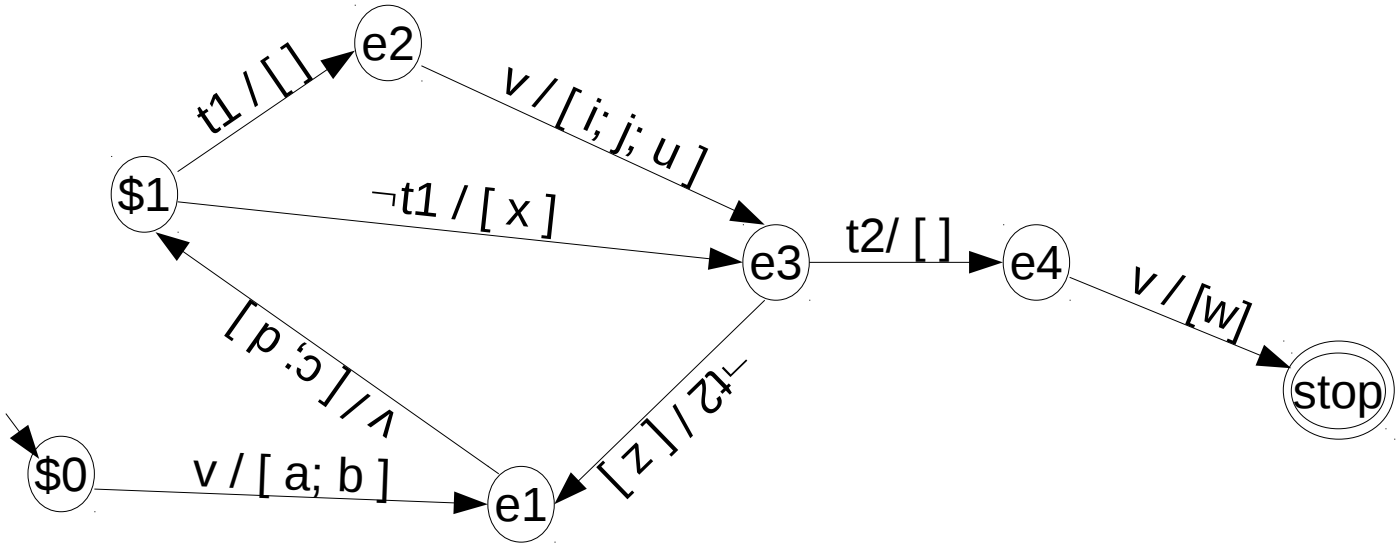
```
$0 :    [a; b] ; goto e1;
e1 :    [c; d] ; goto $1
$1 :    if t1 goto e2 ;
        else [x] ; goto e3 ;
e3 :    if t2 goto e4 ;
        else [z] ; goto e1 ;
e2 :    [i ; j ; u] ; goto e3 ;
e4 :    [w] ; goto stop
```

2



Exemple de transformation par automate : B → D

etat	Cnd	Vrai	Faux
$0	v	[a;b] → e1	-
e1	v	[c;d] → $1	-
$1	t1	→ e2	[x] → e3
e2	v	[i;j;u] → e3	-
e3	t2	→ e4	[z] → e1
e4	v	[c;d] → e1	-



3

```
etat ← $0 ;
while ( etat  !=  stop )
      switch (etat) {
            case $0 : a ; b ; etat ← e1 ; break ;
            case e1 : c ; d ; etat ← $1 ; break ;
            case $1 : if (t1) etat ← e2 ; else x ; etat ← e3 ; break ;
            case e2 : i ; j ; u ; etat ← e3 ; break ;
            case e3 : if (t2) etat ← e4 ; else z ; etat ← e1 ; break ;
            case e4 : w ; etat ← stop ; break ;
      }
```

B. Transformation Böhm et Jacopini

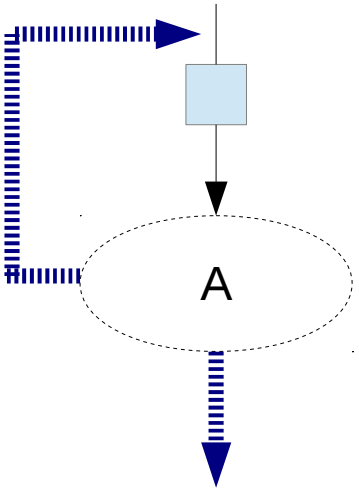
C'est une démonstration du théorème D $=_{FN}$ B

1) D $\leqslant_{FN}$ B (Transformation Triviale)

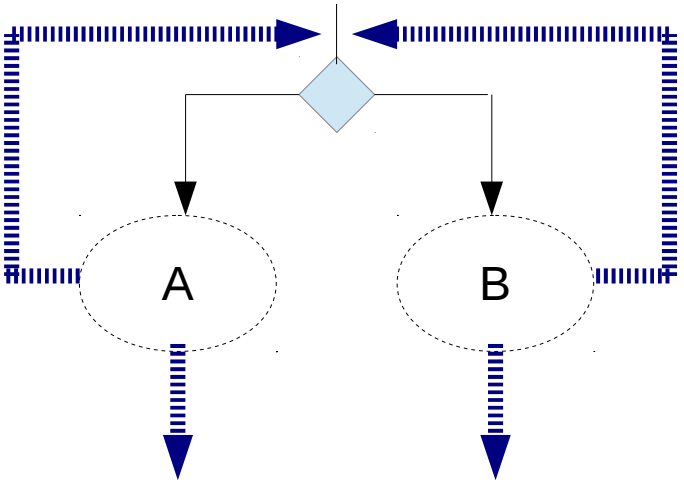
2) B $\leqslant_{FN}$ D :

Tout B-schéma peut se représenter graphiquement à l'aide d'un organigramme.
On montre que tout organigramme (non structuré) peut se transformer en un organigramme structuré fonctionnellement équivalent.
La démonstration se fait par récurrence sur la complexité de l'organigramme.

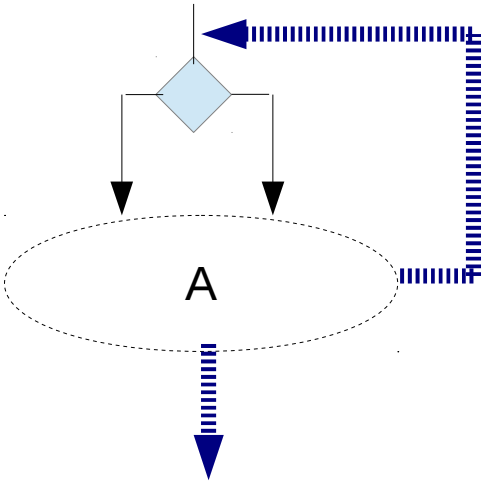
On commence par isoler la première action, cela permet de distinguer 3 types :



Type 1



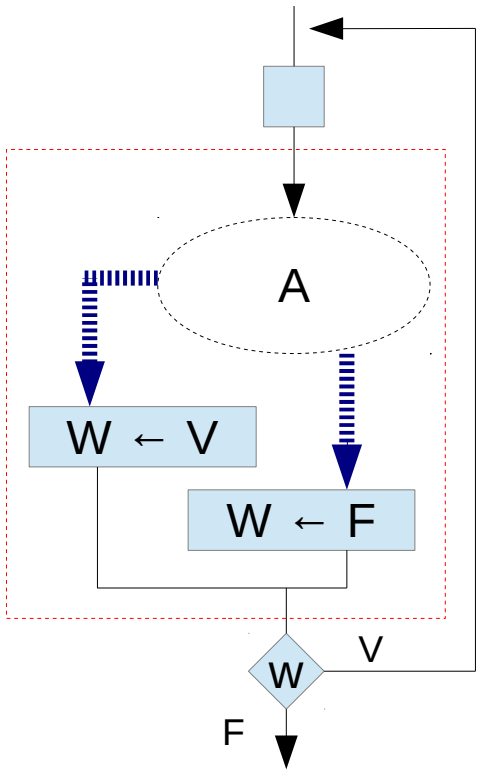
Type 2



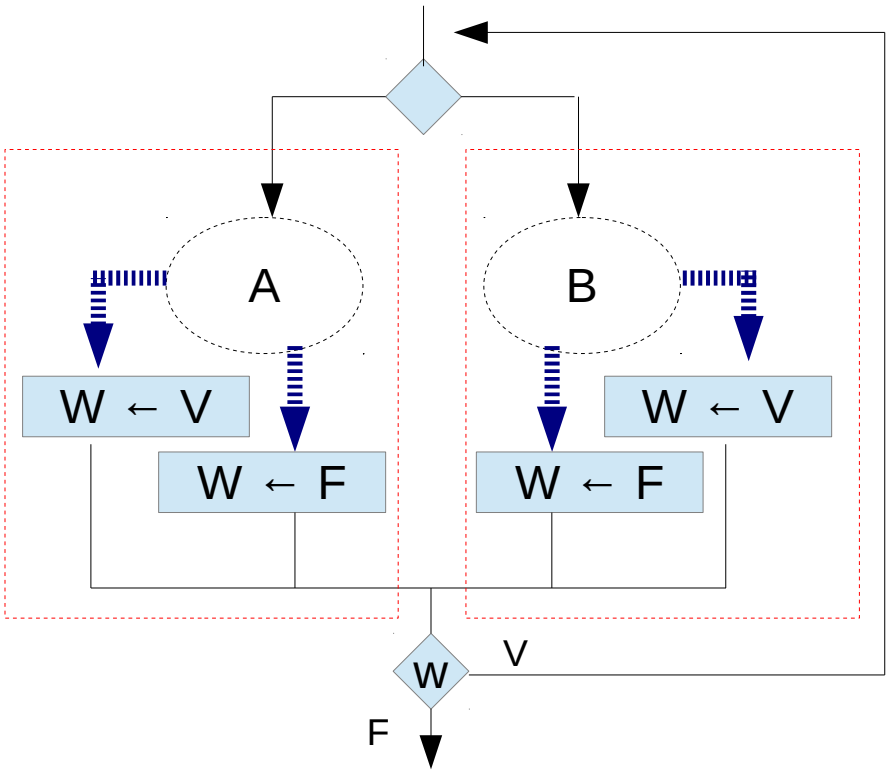
Type 3

Transformer chaque blocs non structuré en boucle 'répéter' en rajoutant des variables booléennes W, des affectations et des tests sur ces variables rajoutées.

Type 1 structuré

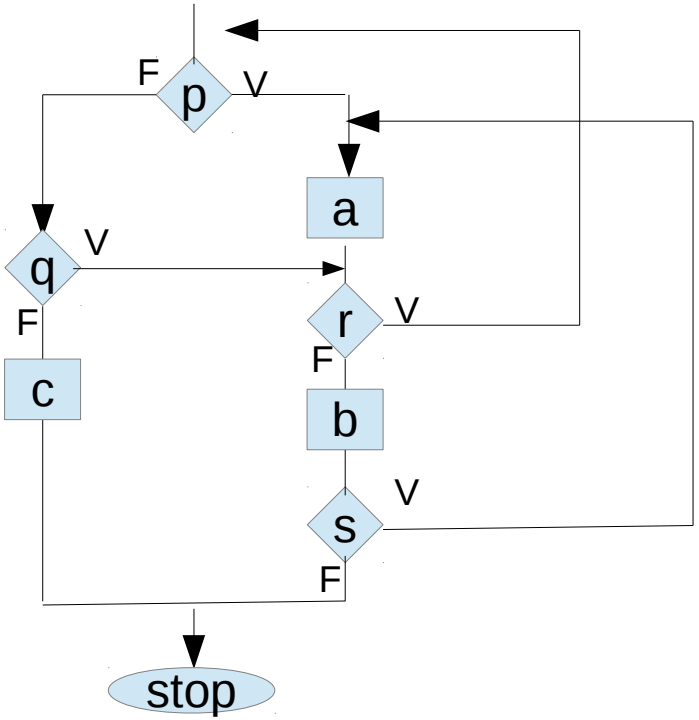


Type 2 structuré

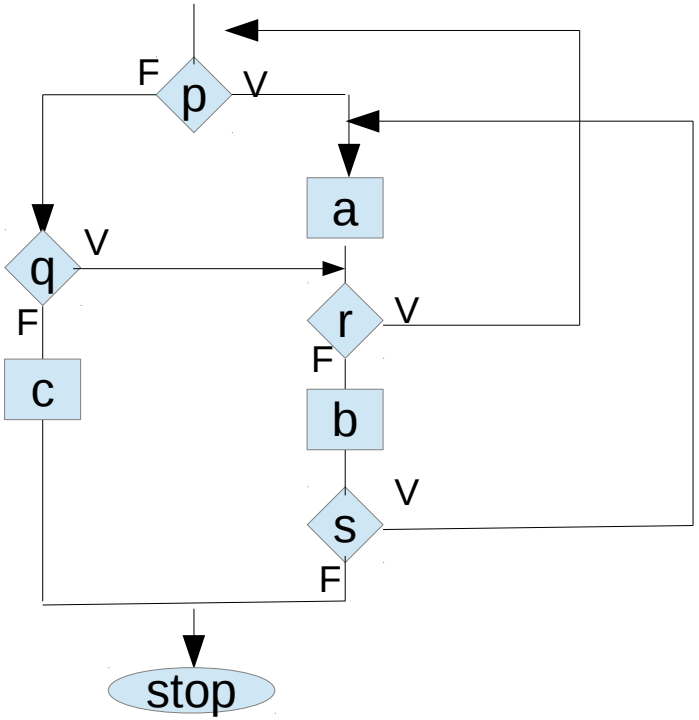


Le type 3 sera d'abord transformé en type2 en dupliquant certaines parties (ou tout) le sous-organigramme A de parts et d'autres du test initial.

La transformation continue avec les sous-organigrammes qui restent (en rouge) en appliquant le même principe, jusqu'à obtenir des organigrammes structurés
A la fin chaque boucle de type 'répéter' sera facilement transformée en 'Tantque' pour avoir un D-schéma.

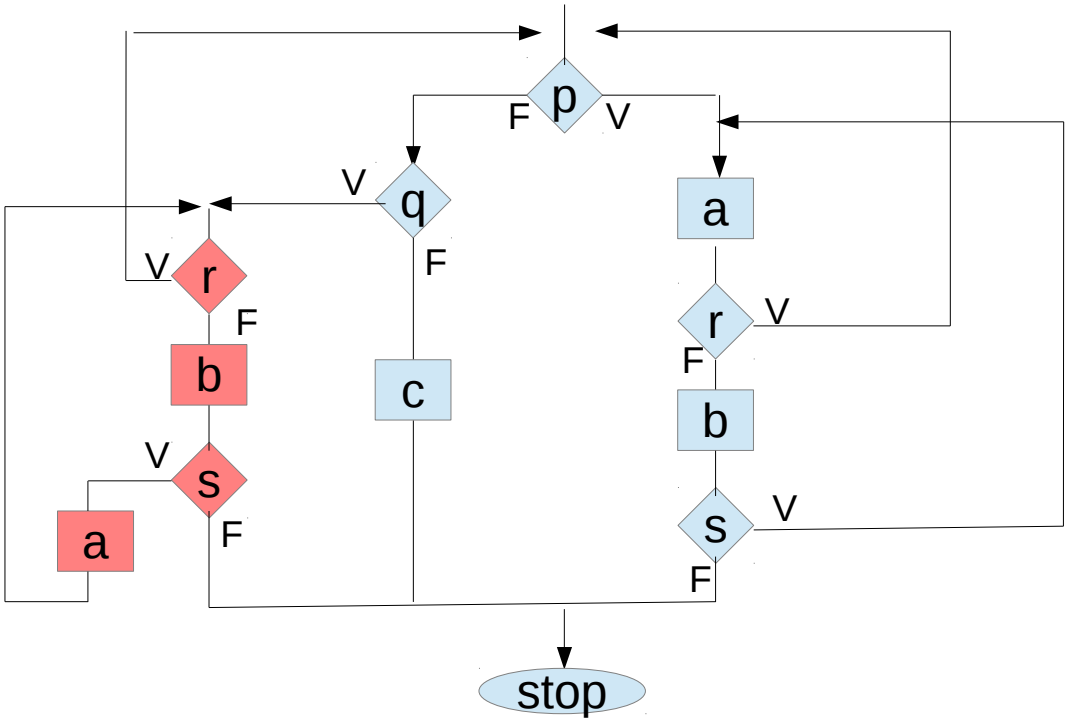


Type 3

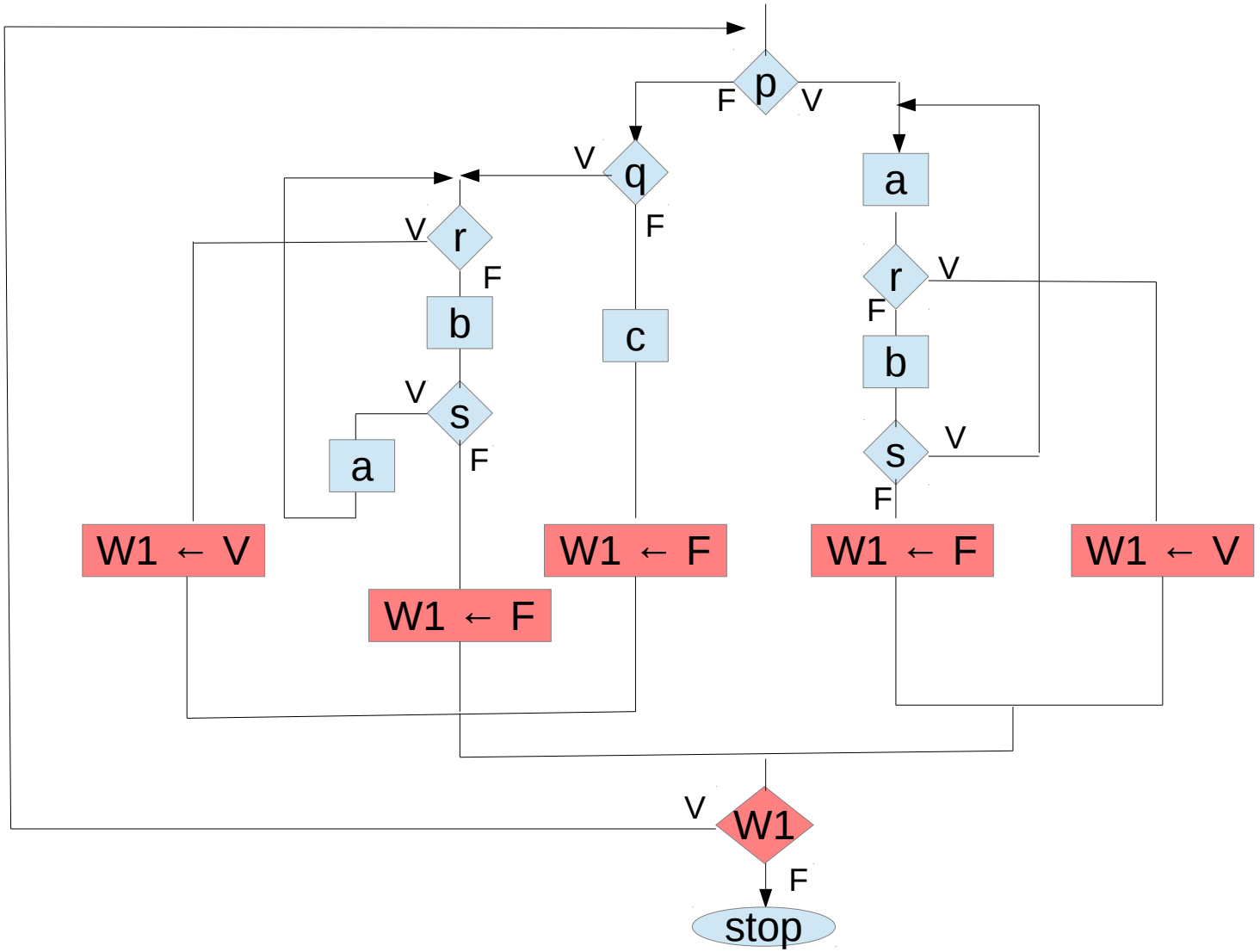


Type 3

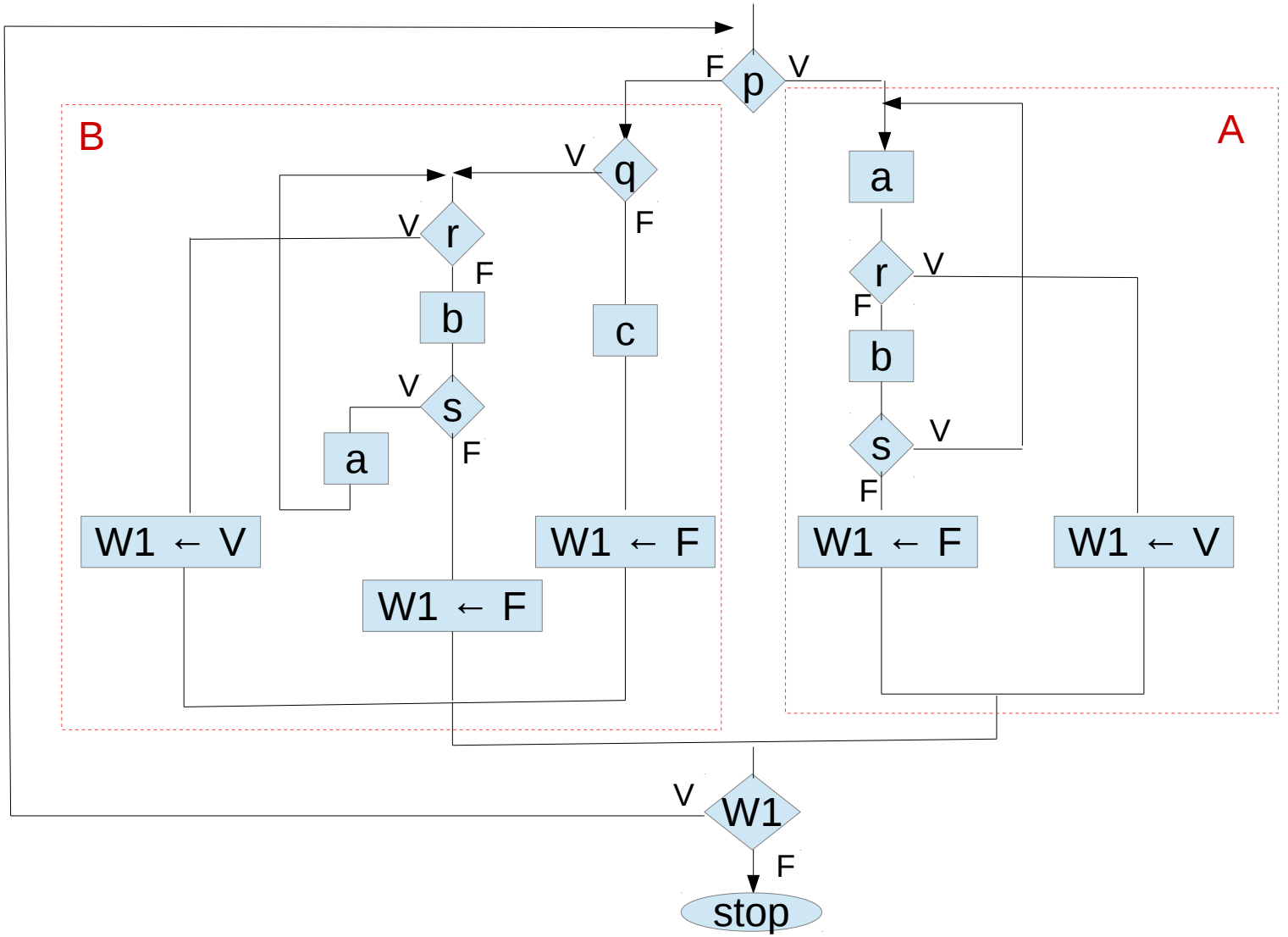
Type 2



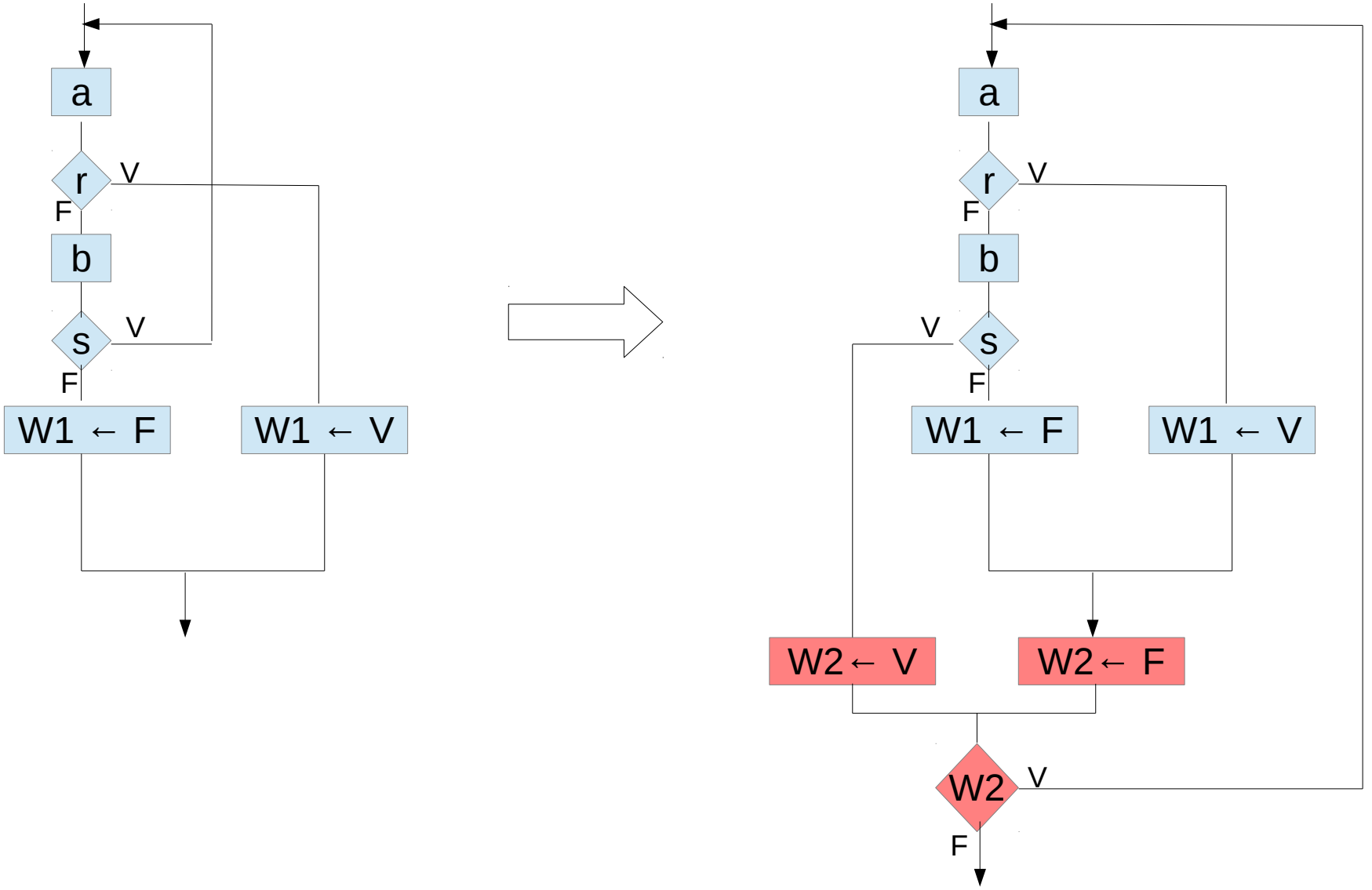
Type 2



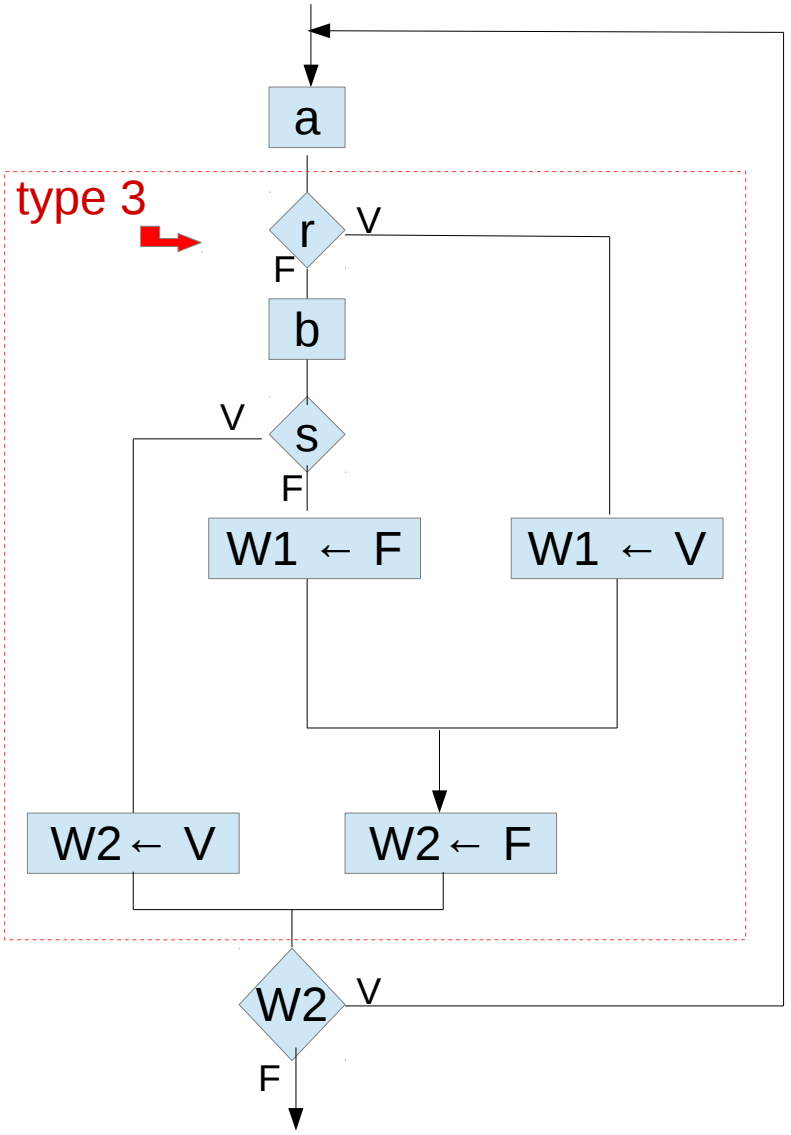
Type 2



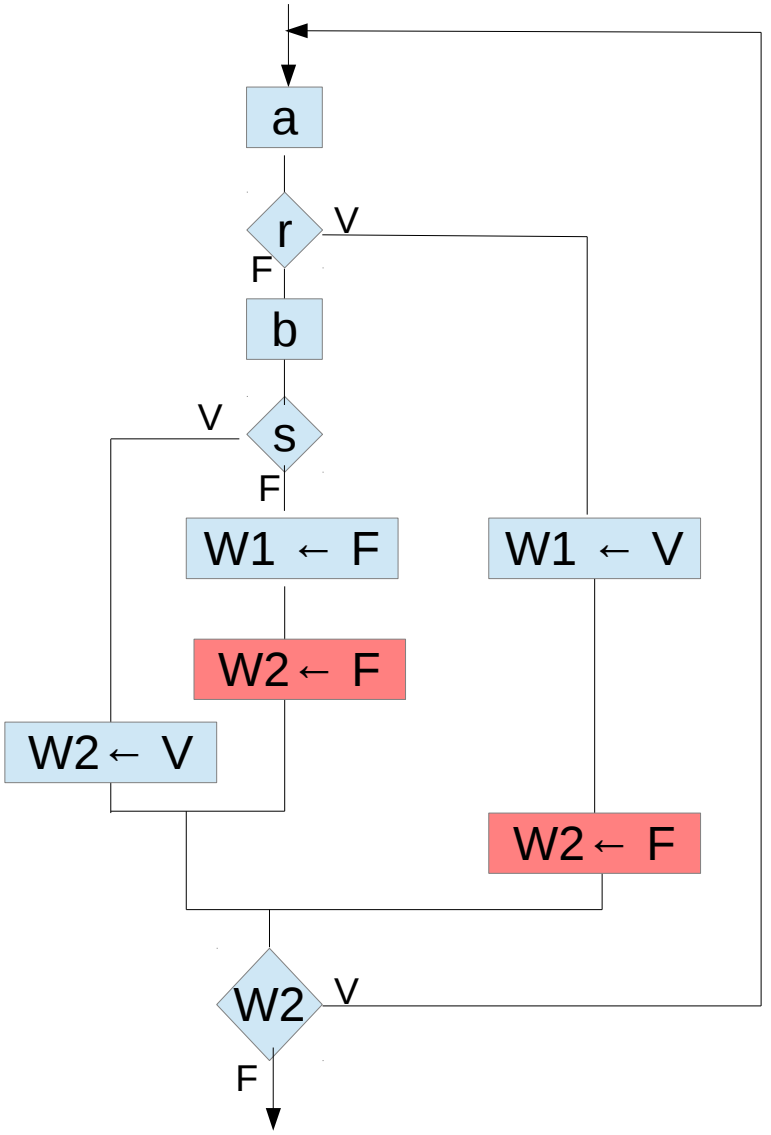
Transformation de A (type 1)



Transformation de A

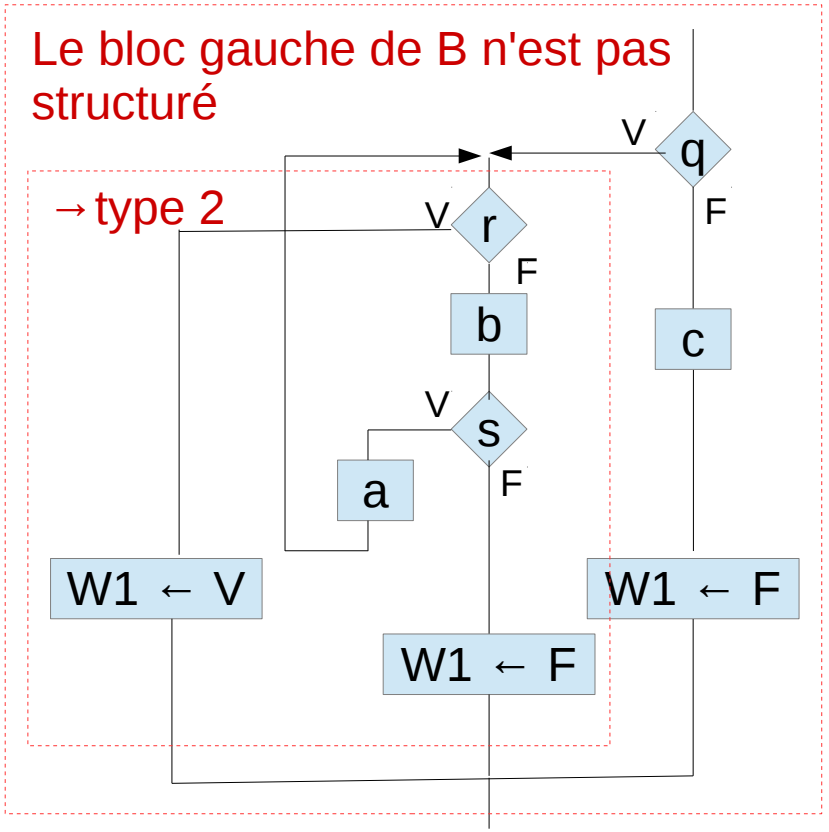


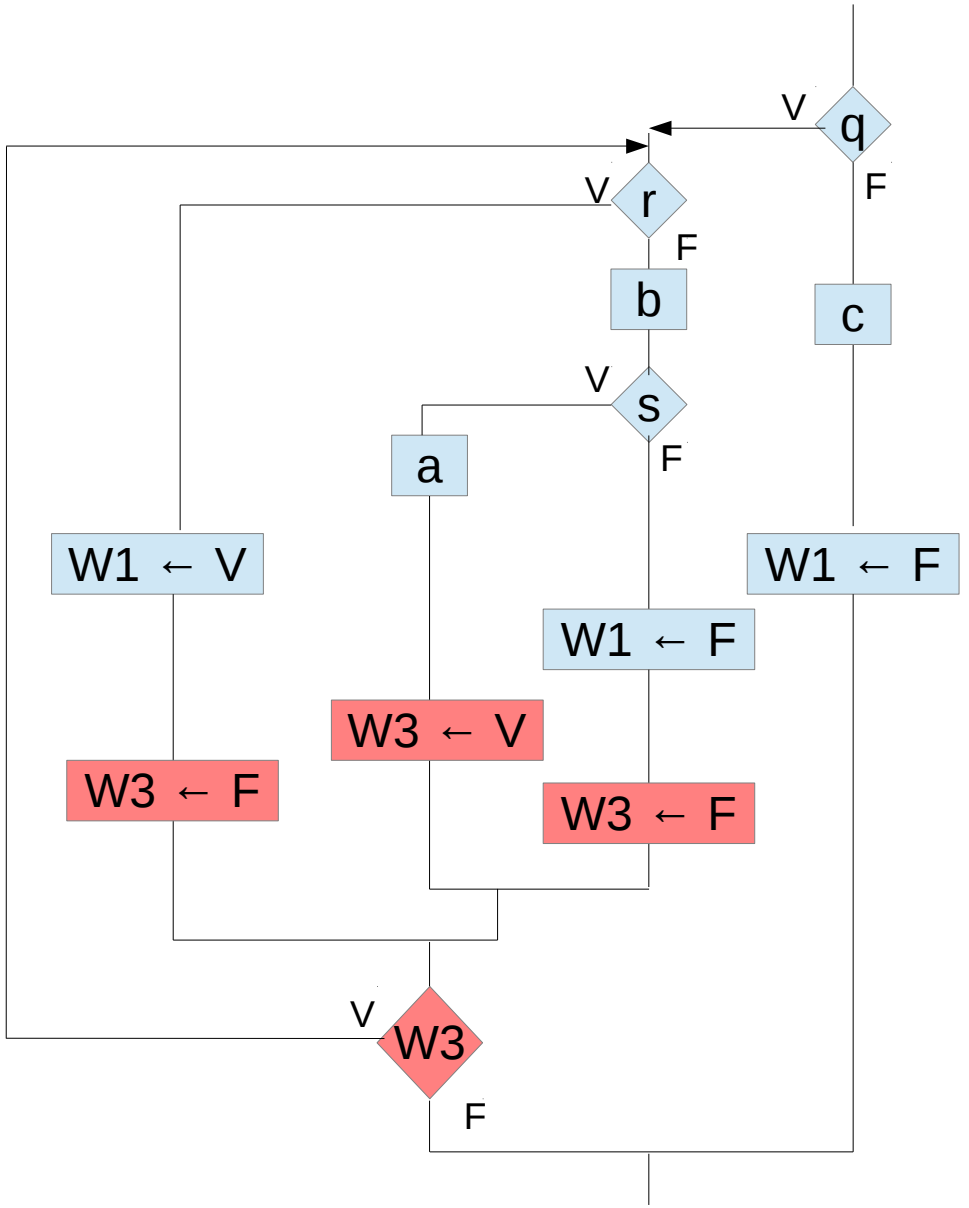
Résultat = A' (structuré)



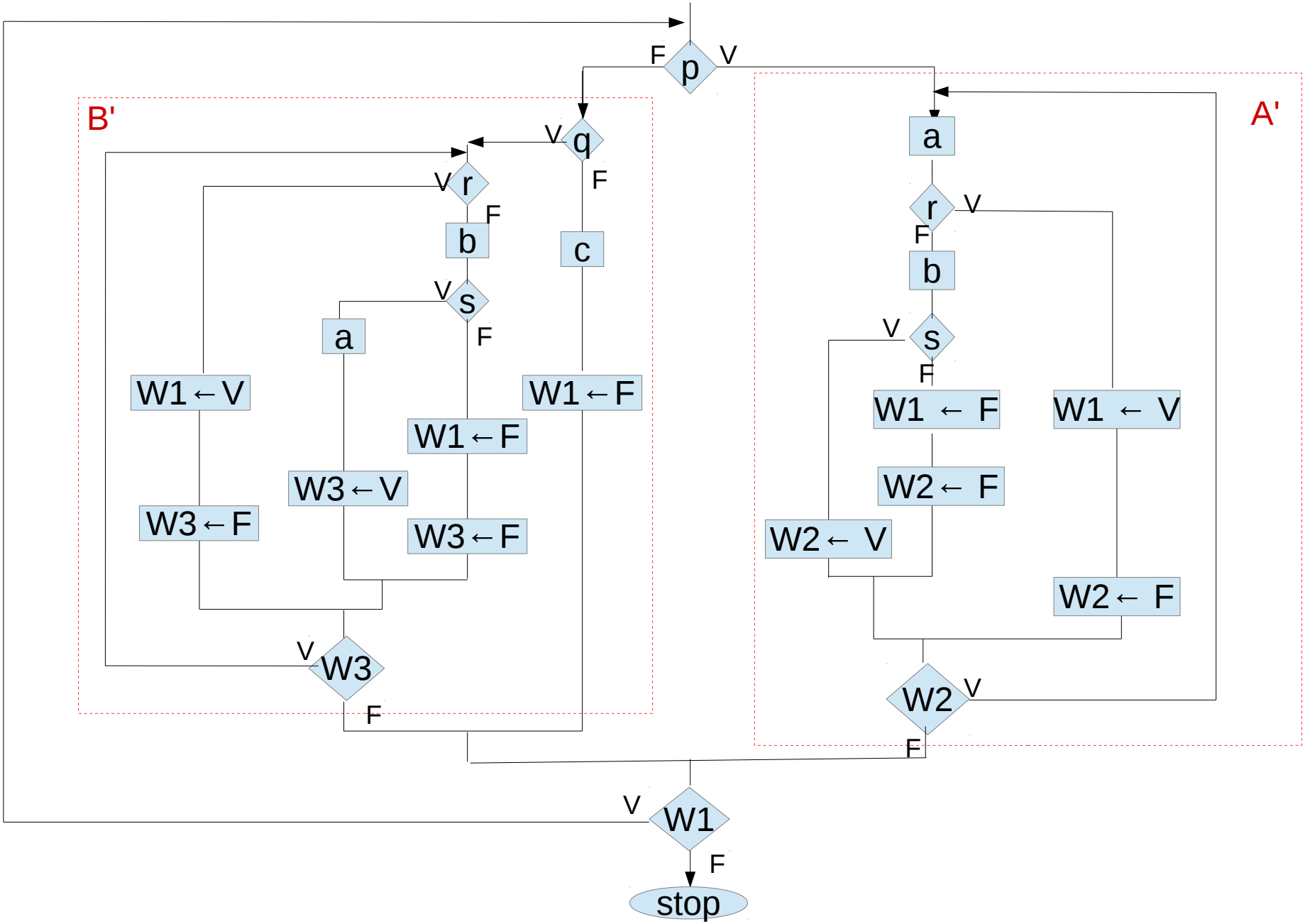
Transformation de B

Résultat = B' (structuré)





Résultat final : organigramme structuré



C. Preuves Formelles

Un programme P **calcule** une fonction f

Prouver P consiste à vérifier :
$\forall$ x $\in$ Domaine de définition de f, la sortie produite par P pour l'entrée x doit être égale à f(x).

Les tests empiriques ne sont pas des preuves formelles.

Plusieurs méthodes de preuves existent en fonction du type de langage procédural visé :

- Méthode de Floyd : pour les B-Schémas
- Méthode de Dijkstra : pour les D-schémas
- Méthode de Hoare : pour les D-Schémas

Les moins difficiles sont celles concernant les schémas les plus structurés (D-Schémas)

C. Système de preuve de HOARE

C'est un système formel basé sur la logique du 1er ordre et les théories manipulées par chaque programme.

- Enoncés de la forme : $E \{ P \} S$

E : Précondition, S : Postcondition, P : Programme

Si les valeurs des variables de P vérifient le prédicat E avant l'exécution de P,
Alors Si P se termine, elles vérifieront le prédicat S

- Axiome de l'affectation (**AFF**) : $E(exp) \{ x \leftarrow exp \} S(x)$

Tout énoncé de cette forme est considéré comme vrai

E(exp) est la plus faible précondition nécessaire pour que S(x) soit vraie après l'affectation. (c-a-d : celle imposant le minimum de contraintes aux variables).

L'espace des conditions munis de la relation d'ordre partielle 'plus faible que' est inductif (c'est même un treillis). Son plus petit élément est la condition 'Vrai' qui n'impose aucune contrainte sur les variables et son plus grand élément est la condition 'Faux' qui impose tellement de contraintes sur les variables qu'aucune valeur ne peut la satisfaire.

Exemples: $(x+y > 3) \{y \leftarrow x+y\} (y > 3)$, $(1=1) \{z \leftarrow 1\} (z=1)$, $(1=2) \{x \leftarrow 1\} (x=2)$

Ces trois énoncés sont vrais car ce sont des axiomes de l'affectation

- Règles d'inférences :

a) Règles de l'implication :

(**IMP1**) $E' \{ P \} S \ , \ E \Rightarrow E' \ |\!-\!- \ E\{P\}S$

(**IMP2**) $E \{ P \} S' \ , \ S' \Rightarrow S \ |\!-\!- \ E\{P\}S$

Exemple, on peut démontrer que (y = 3) { x ← y } (x > 0) à 'aide de IMP1 et AFF

b) Règle de la séquentielle (**SEQ**) : $E \{ P \} F \ , \ F \{ Q \} S \ |\!-\!- \ E \{P;Q\} S$

Exemple :

(y=2) {z ← y ; x ← z} (x=2) est vrai d'après la règle SEQ, car les deux énoncés suivants : (z=2) {x ← z} (x=2) et (y=2) {z ← y}(z=2) sont aussi vrais (axiomes AFF).

c) Règles la conditionnelle :

(**CND1**) : $(E \ \& \ B) \{P\} S \ , (E \ \& \ \neg B) \Rightarrow S \ |\!-\!- \ E \{Si \ (B) \ P \ Fsi\} S$

(**CND2**) : $(E \ \& \ B) \{P\} S \ , (E \ \& \ \neg B) \{Q\} S \ |\!-\!- \ E \{Si \ (B) \ P \ Sinon \ Q \ Fsi\} S$

Exemple :

(x=a & y=b) { Si (x > y) z ← x Sinon z ← y Fsi } (z = max(a,b))

Cet énoncé peut être démontré à l'aide des règles IMP1, CND2 et l'axiome AFF.

d) Règle de l'itération (**ITE**) : $(E \ \& \ B) \{P\} E \ |\!-\!- \ E \{TQ \ (B) \ P \ Ftq\} (E \ \& \ \neg B)$

Le prédicat E est appelé 'Invariant de la boucle' car c'est une condition qui est vraie au début et à la fin de chaque itération (et aussi au début et à la fin de la boucle).

L'un des problèmes majeurs de cette méthode reste la découverte des 'bons' invariants de boucles.

- Exemple 1

Soit DIV le programme suivant :

```
q ← 0 ;  r ← a ;            ]  P

TQ ( r >= b )               ⎤
     q ← q + 1 ;            ⎥  Q
     r ← r – b              ⎥
FTQ                         ⎦
```

Ce programme calcule dans les variables ‘q’ et ‘r’, le quotient et le reste de la division de ‘a’ par ‘b’. Il faut démontrer l’énoncé : **(a ≥ 0 & b > 0) { DIV } (a = bq+r & r<b)**

Posons P = [q ← 0 ; r ← a] et Q = [TQ (r >= b) q ← q + 1 ; r ← r – b FTQ]

Donc prouver **(a ≥ 0 & b > 0) { DIV } (a = bq+r & r<b)**

Revient à prouver **(a ≥ 0 & b > 0) { P ; Q } (a = bq+r & r<b)**

D’après la règle de la séquentielle (SEQ), il suffirait de prouver :
(a) **(a ≥ 0 & b > 0) { P } F** et (b) **F { Q } (a = bq+r & r<b)**

L'énoncé (b) **F { Q } (a = bq+r & r<b)** est de la forme :

F { TQ (r >=b) q ← q + 1 ; r ← r – b FTQ } (a = bq+r & r<b), ou encore :

F { TQ (B) q ← q + 1 ; r ← r – b FTQ } (F & ¬B)
avec B la condition (r ≥ b) et F la condition (a = bq+r)

Donc on peut le prouver en utilisant la règle de l'itération (ITE). Il suffit alors de prouver l'énoncé suivant : (c) **(F & B) { q ← q + 1 ; r ← r – b } F**

En appliquant la règle SEQ, on pourra prouver (c) à condition de montrer les deux énoncés : (d) **(F & B) { q ← q + 1 } G** et (e) **G { r ← r – b } F**

L'énoncé (e) est vrai car c'est un axiome (AFF) en choisissant G : (a = bq+r-b)

L'énoncé (d) est aussi vrai par (IMP1) car d'une part :
(a = b(q+1)+r-b) { q ← q + 1 } G est vrai par (AFF)
et d'autre part l'implication : **(F & B) ⇒ (a = b(q+1)+r-b)** est aussi vraie

L'énoncé (a) **(a ≥ 0 & b > 0) { P } F** est vrai par SEQ car ses deux prémisses :
(f) **(a ≥ 0 & b > 0) { q ← 0 } H** et (g) **H { r ← a } F** le sont aussi
En effet (g) est un axiome (AFF) à condition de choisir H : (a=bq+a)
Alors que (f) est vrai par la règle IMP1 car :
(a =0+a) { q ← 0 } H est un axiome (AFF) et **(a ≥ 0 & b > 0) ⇒ vrai** est une implication toujours vraie.

- Exemple 2

Soit P le programme suivant :

```
x ← 0 ;  y ← 1 ;  z ← 1 ;
TQ ( y <= n )
    x ← x + 1 ;
    y ← y + z + 2 ;
    z ← z + 2
FTQ
```

Ce programme est censé calculer dans la variable x l'entier approximant la racine carrée de n.
Plus formellement on peut spécifier ce programme en donnant sa précondition $E = (n \geq 0)$ et sa postcondition $S = (x^2 \leq n < (x+1)^2)$.
Il s'agit alors de démontrer que l'énoncé suivant : $(n \geq 0)\ \{ P \}\ (x^2 \leq n < (x+1)^2)$ soit un théorème dans le système formel de Hoare.

Soit le découpage de P suivant :
Posons P1 = [x ← 0 ; y ← 1 ; z ← 1] , P2 = [TQ (y <= n) P3 FTQ] et
P3 = [x ← x + 1 ; y ← y + z + 2 ; z ← z + 2]
Donc P peut être vu comme une séquence entre P1 et P2, P = [P1 ; P2]

Ainsi l'énoncé de départ E { P } S serait vrai par la règle SEQ si il existe un prédicat F, /que les deux énoncés suivants sont vrais : (a) E { P1 } F et (b) F { P2 } S

Afin d'imposer le moins de contraintes à la découverte des prédicats internes (tel que F), il est préférable de commencer par démontrer les derniers énoncés du programme en premier. L'axiome de AFF nous permettra alors de générer les plus faibles préconditions comme prédicats internes.

Commençons par prouver l'énoncé (b) F { P2 } S
c-a-d (b) F { TQ (y <= n) P3 FTQ } ($x^2 \leq n < (x+1)^2$),
avec P3 le corps du TQ : P3 = [x ← x + 1 ; y ← y + z + 2 ; z ← z + 2]

Puisque P2 est une instruction de type TQ … FTQ, on utilisera alors la règle ITE.
Cette dernière dit que tout énoncé de la forme E {TQ (B) P FTQ } (E & ¬B)
est vrai, si un énoncé de la forme (E & B) { P } E a déjà été démontré.

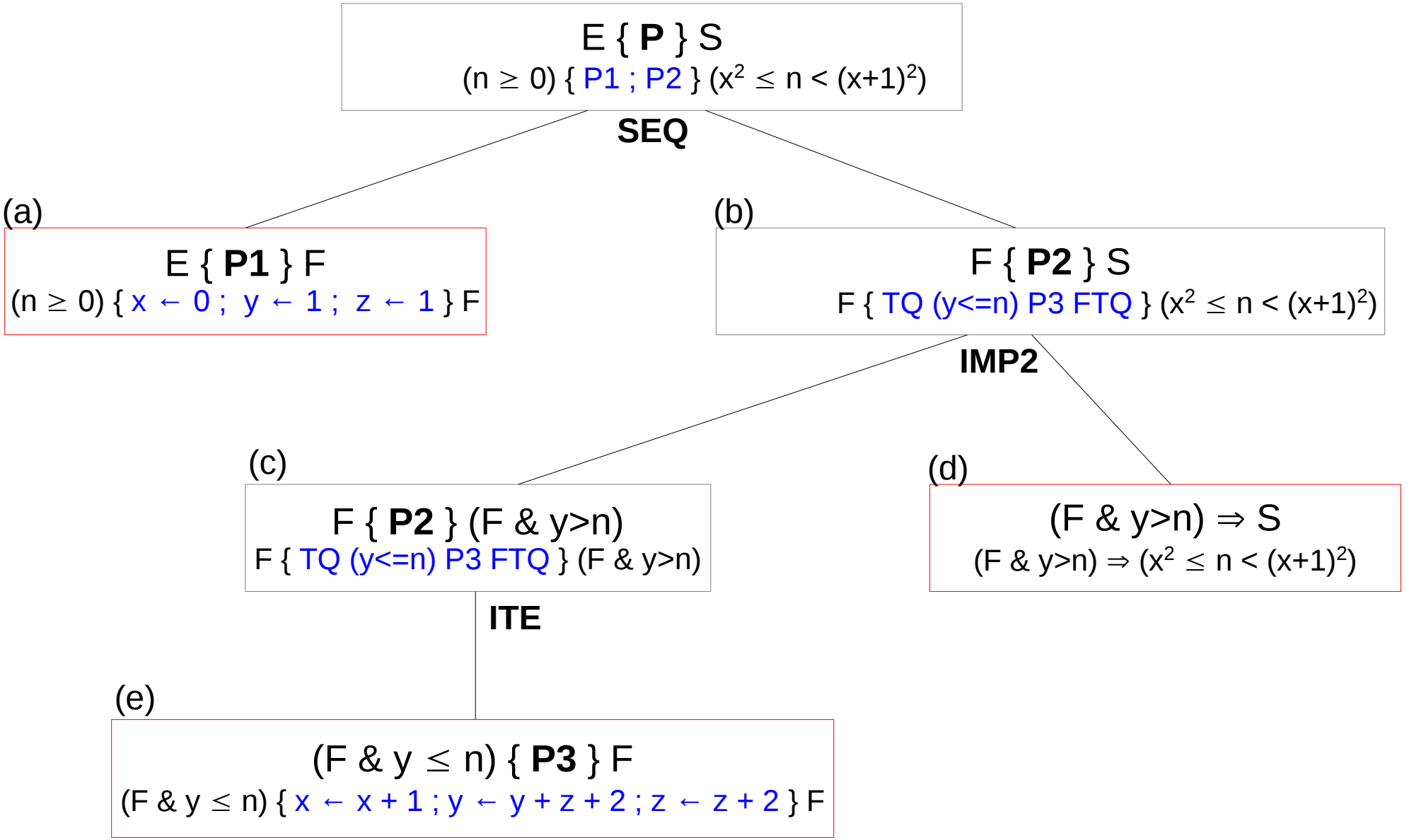
L'énoncé (b) n'a pas exactement la forme d'une conclusion de la règle ITE, donc on ne peut pas l'appliquer directement.

Par contre, en utilisant la règle IMP2, on pourrait connecter (b) avec un énoncé ayant la forme d'une conclusion de ITE, car la différence se situe au niveau de la postcondition :

(b) est vrai par IMP2 si : (c) F { P2 } (F & y>n) et (d) (F & y>n) ⇒ S
ensuite (c) sera vrai par ITE si (e) (F & $y \leq n$) { P3 } F

Remarque : F est le bon invariant à découvrir !

Voici une partie de l'arbre de preuve qui résume ce que l'on vient de faire :



On <u>choisit un bon invariant F</u> et on démontre les énoncés : (d), (e) et (a)

Choix de F vérifiant les 3 énoncés qui restent à prouver

(d) : $(F \;\&\; y>n) \Rightarrow (x^2 \leq n < (x+1)^2)$

(e) : $(F \;\&\; y \leq n)\ \{ x \leftarrow x + 1 ; y \leftarrow y + z + 2 ; z \leftarrow z + 2 \}\ F$

(a) : $(n \geq 0)\ \{ x \leftarrow 0 ;\ y \leftarrow 1 ;\ z \leftarrow 1 \}\ F$

(e) est vrai par SEQ, AFF et IMP1 à condition que :

$F(x,y,z) \;\&\; y \leq n \Rightarrow F(x+1, y+z+2, z+2)$

car $F(x+1, y+z+2, z+2)$ est la plus faible précondition avant $x \leftarrow x + 1$

(a) est vrai par SEQ, AFF et IMP1 à condition que :

$(n \geq 0) \Rightarrow F(0,1,1)$ car $F(0,1,1)$ est la plus faible précondition avant $x \leftarrow 0$

Donc le bon invariant F doit vérifier ces trois implications en même temps :

1. $(F(x,y,z) \;\&\; y>n) \Rightarrow (x^2 \leq n < (x+1)^2)$
2. $F(x,y,z) \;\&\; y \leq n \Rightarrow F(x+1, y+z+2, z+2)$
3. $(n \geq 0) \Rightarrow F(0,1,1)$

D'après la 1ere implication, on peut supposer que F renferme la condition : $y=(x+1)^2$, car dans l'antécédent nous avons que $y>n$ et dans le conséquent nous avons $(x+1)^2>n$. La condition qui reste alors et que $x^2 \leq n$.

Première proposition F : $(x^2 \leq n \;\&\; y=(x+1)^2)$

satisfait 1. mais pas 2. (il manque une relation sur z)

en explicitant l'implication 2 avec le F proposé, on remarque que la relation qui manque est : $z =2x+1$

Deuxième proposition (finale) **F : $(x^2 \leq n \;\&\; y=(x+1)^2 \;\&\; z =2x+1)$**

1. 2. et 3. sont vérifiées, donc c'est le bon invariant.

C. Extension aux appels récursifs

On peut étendre le système de preuve de Hoare pour la prise en compte des appels de procédures et les programmes récursifs

/* programme appelant … */
…
…
Proc(exp_1, exp_2, … exp_n, z_1, z_2, ...z_m)
…
…
…

/* Procédure appelée … */
Proc(x_1,x_2, ...x_n, y_1, y_2, ...y_m)
Entrées : x_1,x_2, ...x_n
Sorties : y_1, y_2, ...y_m
Corps P
...
Fin_Proc

- Règle de l'appel (**APP**) :

$$E\ (x_1,...x_n)\{\ P\ \}\ S(x_1,...x_n,\ y_1,...y_m\ )\ \ |\text{--}$$
$$E\ (exp_1,...exp_n)\{\ Proc(exp_1,...exp_n,\ z_1,...z_m\ )\ \}\ S(exp_1,...exp_n,\ z_1,...z_m\ )$$

Si le corps P est vrai par rapport à ses prédicats d'entrée E et de sortie S,
Alors tout appel à cette procédure (Proc) sera aussi vrai par rapport aux même prédicats E et S mais appliqués aux paramètres effectifs

- En cas d'appels récursifs la règle APP provoque une inférence qui ne termine pas, on utilise alors le théorème suivant :
Si en prenant comme hypothèse que les appels internes sont corrects, on arrive à prouver le corps, lors la procédure récursive est effectivement correcte.

Exemple de preuve par la méthode de Hoare d'une procédure récursive :

Div(a,b,q,r) Début du corps

Entrées : a,b SI (a < b) q ← 0 ; r ← a Sinon Div(a-b, b, q, r) ; q ← q+1 Fsi

Sorties : q,r Fin du corps