Correction 24-25 s2 | Compilation

2025-12-26

F. B
Universite de Montpellier
2025
M1

Contents

I. Exercice 1 . . . 3
I.1. Que fait f . . . 3
I.2. PP . . . 3
I.3. UPP . . . 3
I.4. RTL . . . 3
I.5. ERTL . . . 4
II. Exercice 2 . . . 5
II.1. . . . 5
II.2. . . . 6
III. Exercice 3 . . . 7
III.1. . . . 7
III.2. . . . 7
III.3. . . . 8
IV. Exercice 4 . . . 11

I. Exercice 1

I.1. Que fait f

f(5) = f(4) + 2 = f(3) + 4 = f(2) + 6 = f(1) + 8 = f(0) + 10 = 10
La fonction va calculer le double de son entree

I.2. PP

PP

```
function f(n: integer): integer
  if n = 0 then
      return 0
  else
      return f(n - 1) + 2
```

I.3. UPP

UPP = PP sans typage

UPP

```
function f(n) :
  if n = 0 then
    return 0
  else
    return f(n-1) + 2
```

I.4. RTL

RTL

```
function f(%0) %1
var %0 %1 %2
entry start
exit final

start: beqz %0 -> base, rec     ; Si %0 == 0 on va à 'base', sinon 'rec'

base:  li %1 0 -> final         ; Cas de base : résultat = 0, et on sort
vers 'final'

rec:   sub %2, %0, 1 -> call    ; Calcul n-1
call:  call %1 f(%2) -> add     ; Appel f(n-1)
add:   add %1, %1, 2 -> final   ; Résultat + 2, et on sort vers 'final'

final:
```

I.5. ERTL

ÈRTL

```
1  procedure f(1)
2  var %0 %1 %2 %3
3  entry f0
4  exit f10
5
6  ; --- PROLOGUE ---
7  f0: newframe -> f1
8  f1: move %3 $ra -> f2        ; Sauvegarde de l'adresse de retour
   (CRUCIAL)
9  f2: move %0 $a0 -> f3        ; On récupère l'argument n (depuis le
   registre physique $a0)
10
11 ; --- TEST ---
12 f3: beqz %0 -> f4, f5        ; Si n == 0 -> f4, Sinon -> f5
13
14 ; --- CAS DE BASE (n=0) ---
15 f4: li %1 0 -> f9            ; Résultat = 0, on file vers la fin (f9)
16
17 ; --- CAS RÉCURSIF (n-1) ---
18 f5: addiu %2, %0, -1 -> f6   ; Calcul de n - 1 stocké dans %2
19 f6: move $a0, %2 -> f7       ; On prépare l'argument pour l'appel (dans
   $a0)
20 f7: call f(1) -> f8          ; Appel récursif (écrase $ra, d'où la
   sauvegarde en f1)
21 f8: move %1 $v0 -> f8_bis    ; On récupère le résultat de f(n-1) depuis
   $v0
22
23 ; --- CALCUL FINAL (+2) ---
24 f8_bis: add %1, %1, 2 -> f9  ; On ajoute 2 au résultat
25
26 ; --- ÉPILOGUE (Restauration) ---
27 f9:  move $v0 %1 -> f9_bis   ; On place le résultat final dans le
   registre de retour $v0
28 f9_bis: move $ra %3 -> f9_ter; On restaure l'adresse de retour originale
29 f9_ter: delframe -> f9_quat
30 f9_quat: jr $ra -> f10       ; Saut au registre retour (Return réel)
31
32 f10:                         ; Fin du graphe
```

Un registre callee-save est nécessaire lorsqu'une variable locale doit survivre à travers un appel de fonction (c'est-à-dire qu'elle est définie avant l'appel et utilisée après l'appel).

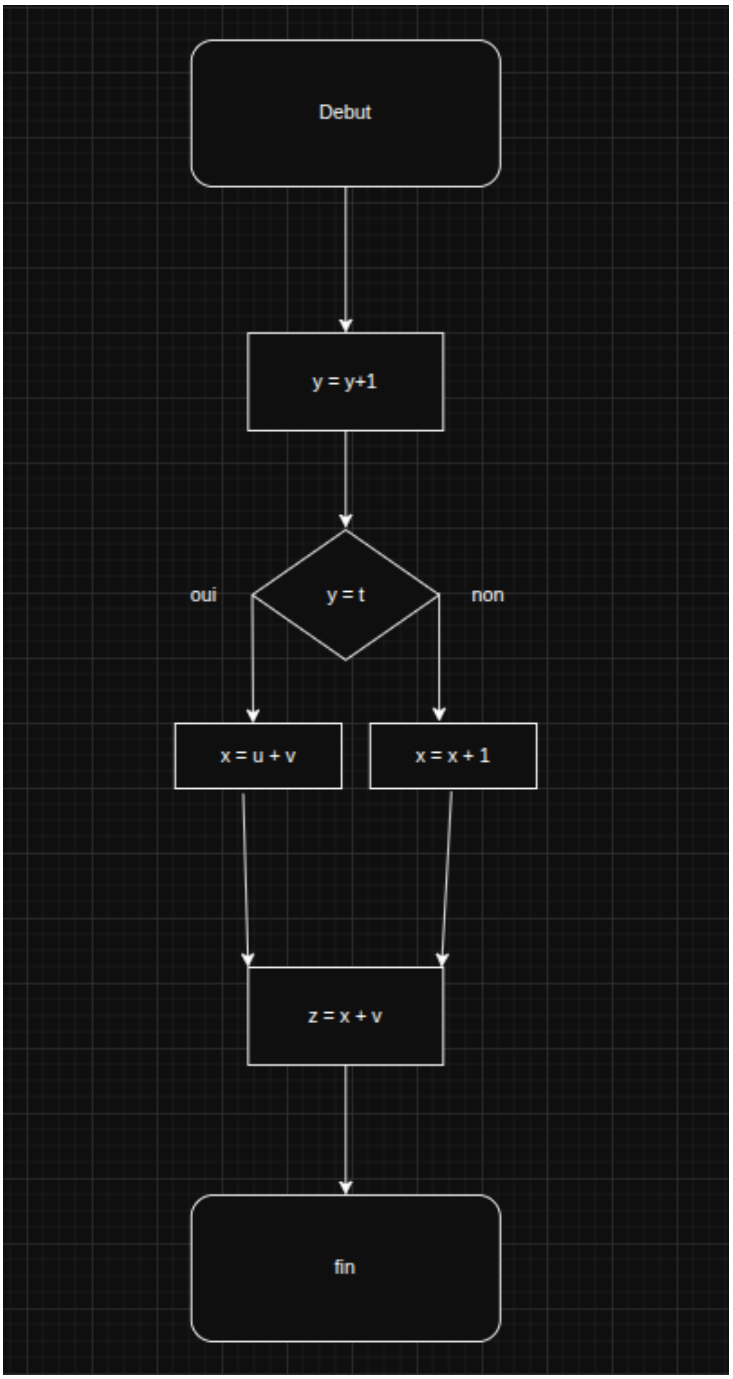
La valeur de n (stockée dans r_1 dans notre RTL) n'est plus jamais utilisée après l'appel à f(n-1). Le calcul final (+ 2) se fait sur le résultat retourné par la fonction, pas sur n.

donc non

II. Exercice 2

II.1.

Le flot de controle :

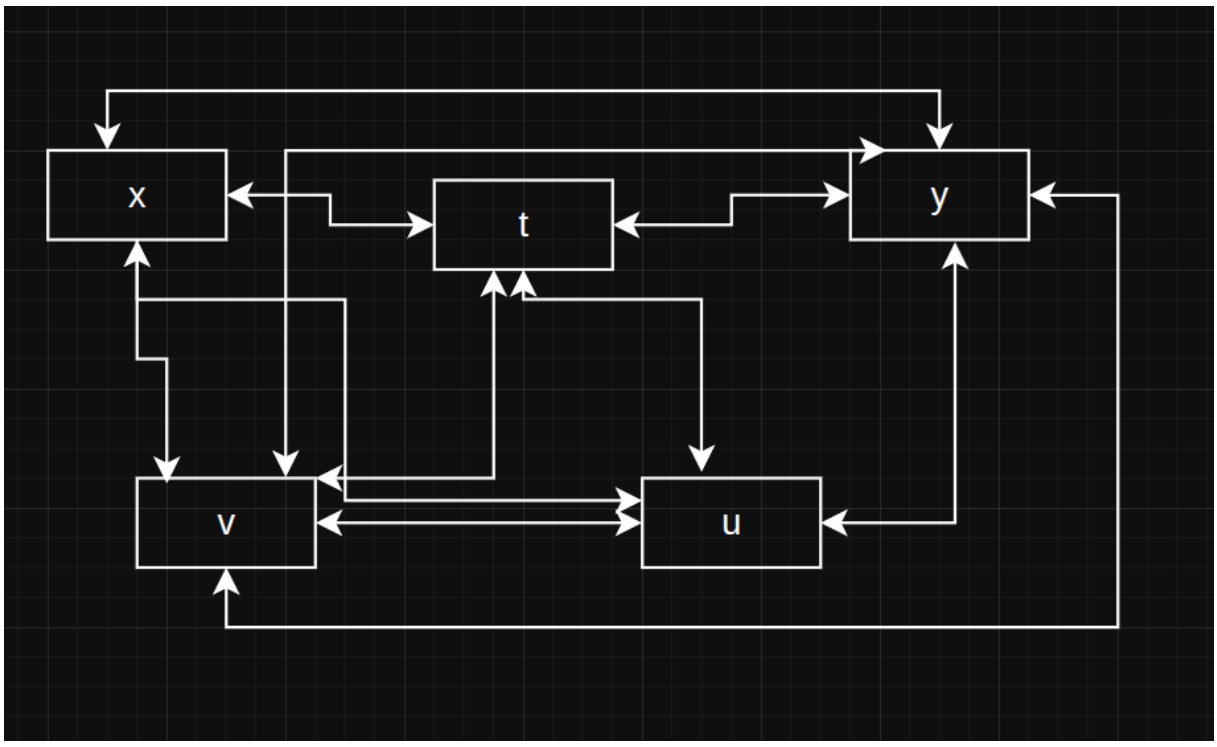


La dure de vie :

```
1 {u, v, t, y, x}
2 y = y + 1 {u, v, t, y, x}
3 if y = t then {u, v, t, y, x}
4   x = u + v {u , v}
5 else
6   x = x + 1 {x,v}
7 z = x + v {x, v}
8 {}
```

Vu que x est dans le if et dans le if il est a la fois lu et ecrase, vu que x = x + 1 cvd que x est lu au debut du prog et il existe et pas nouveau, si c etai x = 1 + 2 alors on l'aurai pas mis au debut

Graphe d'interferance :
x – y , x – t , x – u , x – v , y – t , y – u, y – v , t – u , t – v , u – v
En gros, tt ce qui est au debut on les lie les un au autres, c les variables en entree, ensuite pour chaque definition de variable on le lie au variable vivante de la ligne juste apres.



II.2.

On a besoin de 5 couleurs pour colorier le graphe, vu que on a u, v, t, x, y qui sont lie entre eux.
Pour justifier on cherche un sommet qui a moins que 5 noeud lie a lui. On prend x, on le retire :
y – t , y – u, y – v , t – u , t – v , u – v
On repete jusqua vide donc ca marche avec k = 5 sans devoir spille.

Mais pour k = 4 ca marchera pas, on sera oblige de spille.

III. Exercice 3

III.1.

Les états d un automate peuvent-être traduis au sein de la VM par des labels. Chaque état aura donc son etiquette et il sera possible de sauter à un label pour effectuer une transition. Pour chaque état nous chargerons le caractère correspondant

III.2.

```lisp
(vm-automate '(
  (LABEL start)
  (JMP etat0)          ; L'état initial est 0

  ;; --- ÉTAT 0 (Non Final) ---
  (LABEL etat0)
  (MOVE (:CONST etat0) R2) ; (Optionnel) Pour suivre l'état courant dans R2
  (BNULL R0 refuser)  ; Si la liste R0 est vide, on rejette (0 n'est pas final)
  (CAR R0 R1)          ; R1 prend la valeur du caractère courant (tête de liste)
  (CDR R0 R0)          ; R0 avance au caractère suivant (queue de liste)
  (CMP R1 (:CONST a)) ; Compare le caractère lu avec 'a'
  (JEQ etat1)          ; Si égal, transition vers l'état 1
  (CMP R1 (:CONST c)) ; Compare avec 'c'
  (JEQ etat2)          ; Si égal, transition vers l'état 2
  (JMP refuser)        ; Si c'est autre chose (ex: 'b'), transition invalide -> rejet

  ;; --- ÉTAT 1 (Non Final) ---
  (LABEL etat1)
  (MOVE (:CONST etat1) R2) ; Correction syntaxe: retrait de la parenthèse en trop ')'
  (BNULL R0 refuser)  ; Si vide, on rejette (1 n'est pas final)
  (CAR R0 R1)
  (CDR R0 R0)
  (CMP R1 (:CONST b))
  (JEQ etat1)          ; Boucle sur 'b'
  (CMP R1 (:CONST c)) ; Transition a | c vers l'état 3
  (JEQ etat3)
  (CMP R1 (:CONST a))
  (JEQ etat3)
  (JMP refuser)

```

```
31    ;; --- ÉTAT 2 (Non Final) ---
32    (LABEL etat2)
33    (MOVE (:CONST etat2) R2)
34    (BNULL R0 refuser)  ; Si vide, on rejette
35    (JMP refuser)
36
37    ;; --- ETAT 3 (Final) ---
38    (LABEL etat3)
39    (MOVE (:CONST etat3) R2)
40    (BNULL R0 accepter) ; Si la liste est vide ici (État final)
41    (CAR R0 R1)
42    (CDR R0 R0)
43    (CMP R1 (:CONST a)) ; Transition a | b vers l'état 2
44    (JEQ etat2)
45    (CMP R1 (:CONST b))
46    (JEQ etat2)
47    (CMP R1 (:CONST c)) ; Boucle sur 'c'
48    (JEQ etat3)
49    (JMP refuser)
50
51    ;; --- BLOCS DE FIN ---
52    (LABEL accepter)
53    (MOVE R2 R0) ; On retourne le chemin
54    (HALT)                  ; Arrêt de la machine
55
56    (LABEL refuser)
57    (MOVE (:CONST nil) R0)  ; On retourne Faux/Nil
58    (HALT)
59  ))
```

III.3.

1. Le principe de la génération repose sur la linéarisation du graphe de l'automate. Chaque élément structurel de l'automate est traduit en une séquence d'instructions de la machine virtuelle (VM) selon les règles suivantes :

    1. Les États (Label) : Chaque état de l'automate est traduit par une étiquette (LABEL) unique dans le code (ex: LABEL etat1). Cela définit un point d'ancrage vers lequel on pourra effectuer des sauts (JMP).

    2. L'État Initial (Start) : Le point d'entrée du programme est marqué par un label global (start), suivi immédiatement d'un saut inconditionnel (JMP) vers le label correspondant à l'état initial de l'automate.

    3. La Condition d'Arrêt et les États Finaux : Au tout début du code de chaque état, on vérifie si la chaîne d'entrée est vide (BNULL) :

- Si la chaîne est vide et que l'état est final : on saute vers un bloc accepter (succès).
- Si la chaîne est vide et que l'état est non-final : on saute vers un bloc refuser (échec).

4. Les Transitions (Comparaison + Saut) : Si la chaîne n'est pas vide, on consomme le premier caractère (instructions CAR puis CDR). Ensuite, pour chaque transition sortante de l'état :
   - On compare (CMP) le caractère lu avec le symbole de la transition.
   - Si cela correspond, on effectue un saut conditionnel (JEQ) vers le label de l'état cible.
5. Cas d'erreur (Transition manquante) : Si aucune des comparaisons de transition ne correspond au caractère lu, on place un saut inconditionnel (JMP refuser) à la fin du bloc de l'état pour rejeter le mot.

2. Commençons par définir une fonction faisant les transitions, ici c'est l'élément le plus simple, il s'agit d'une comparaison suivie d'un JEQ. Cette fonction génère le petit bloc d'assembleur qui teste si le caractère lu (R1) correspond à une transition précise.

```
1 (defun gen-auto-transition (symbole-transition etat-suivant)
2   (list
3     ;; On compare le caractère lu (stocké dans R1) avec le symbole attendu
4     `(CMP R1 (:CONST ,symbole-transition))
5     ;; Si c'est égal, on branche directement vers l'étiquette de l'état
      suivant
6     `(JEQ ,etat-suivant)))
7 ))
```

Maintenant, il nous faut-être capable de convertir un état vers une liste d'instruction. C'est le cœur du compilateur. Pour un état donné, on génère le code qui lit le caractère, gère l'arrêt si la chaîne est vide, ou tente de prendre une transition.

```
1  (defun gen-etat-automate (etat automate)
2    (let ((transitions (auto-trans-list automate etat))) ; Récupère les
     transitions de l'état
3      (append
4        ;; 1. En-tête de l'état et lecture
5        `((LABEL ,etat)            ; Étiquette pour les sauts (JMP/JEQ)
         vers cet état
6          (CAR R0 R1)              ; Lecture : Tête de la liste
           (caractère) -> R1
7          (CDR R0 R0)              ; Avance : Queue de la liste (reste) ->
           R0
8          (MOVE (:CONST ,etat) R2)  ; Debug : On note l'état courant dans
           R2
```

```
9
10          ;; 2. Gestion de la fin de chaîne (si R0 est vide/NULL)
11          ,(if (auto-final-p automate etat)
12               '(BNULL accepter)    ; Si fin de chaîne et état final -> SUCCÈS
13               '(BNULL refuser)))   ; Si fin de chaîne et état non final -> ÉCHEC
14
15        ;; 3. Génération des tests pour chaque transition possible
16        ;; On map la fonction précédente sur toutes les transitions de cet état
17        (mapcan (lambda (transition)
18                  (gen-auto-transition (first transition) (second transition)))
19                transitions)
20
21        ;; 4. Cas par défaut
22        '((JMP refuser)))))         ; Si aucun caractère ne matche, le mot est rejeté
```

Et enfin, pour finir, la génération de l'automate au complet. Cette fonction assemble le tout : le point d'entrée, tous les blocs d'états, et les blocs de fin (accepter/refuser).

```
1   (defun auto2vm (automate)
2     (let ((etats (auto-etat-liste automate))   ; Liste de tous les états
3           (etat-initial (auto-init automate))) ; L'état de départ
4       (append
5         ;; --- PROLOGUE ---
6         `((LABEL start)        ; Point d'entrée global du programme
7           (JMP ,etat-initial)) ; Saut immédiat vers l'état initial
8
9         ;; --- CORPS (Génération des états) ---
10        ;; On concatène le code généré pour chaque état de l'automate
11        (mapcan (lambda (etat)
12                  (gen-etat-automate etat automate))
13                etats)
14
15        ;; --- ÉPILOGUE (Blocs de fin) ---
16        '((LABEL accepter)
17          (MOVE (:CONST true) R0) ; Succès : on retourne Vrai (ou le reste de la liste)
18          (HALT)                  ; Arrêt de la VM
19
20          (LABEL refuser)
```

```
21          (MOVE (:CONST nil) R0)  ; Échec : on retourne Nil
22          (HALT)))))              ; Arrêt de la VM
```

IV. Exercice 4