Annales de compilation

VIAL Sébastien

11 septembre 2024

1 Examen de 2023, session 1

Exercice 1 (6 points) :
Soit les fonctions *even* et *odd*, qui testent respectivement si un entier naturel est pair ou impair. Elles peuvent-être écrites en utilisant une récursion mutuelle de la façon suivante :

$$\text{even}(n) = \begin{cases} \top & \text{si } n = 0 \\ odd(n-1) & \text{sinon} \end{cases}$$

$$\text{odd}(n) = \begin{cases} \bot & \text{si } n = 0 \\ even(n-1) & \text{sinon} \end{cases}$$

1. Ecrire les fonctions *even* et *odd* en PP.
2. Traduire les fonctions *even* et *odd* en UPP.
3. Traduire les fonctions *even* et *odd* en RTL.
4. Traduire les fonctions *even* et *odd* en ERTL.
   A-t-on besoin de registres *callee-save* dans cette traduction ? Justifier pourquoi.

1. La traduction en PP est la suivante :

```
function even(n: integer): boolean
    if n = 0 then
        even := true
    else
        even := odd(n-1)

function odd(n: integer): boolean
    if n = 0 then
        odd := false
    else
        odd := even(n-1)
```

2. La traduction en UPP est la suivante :

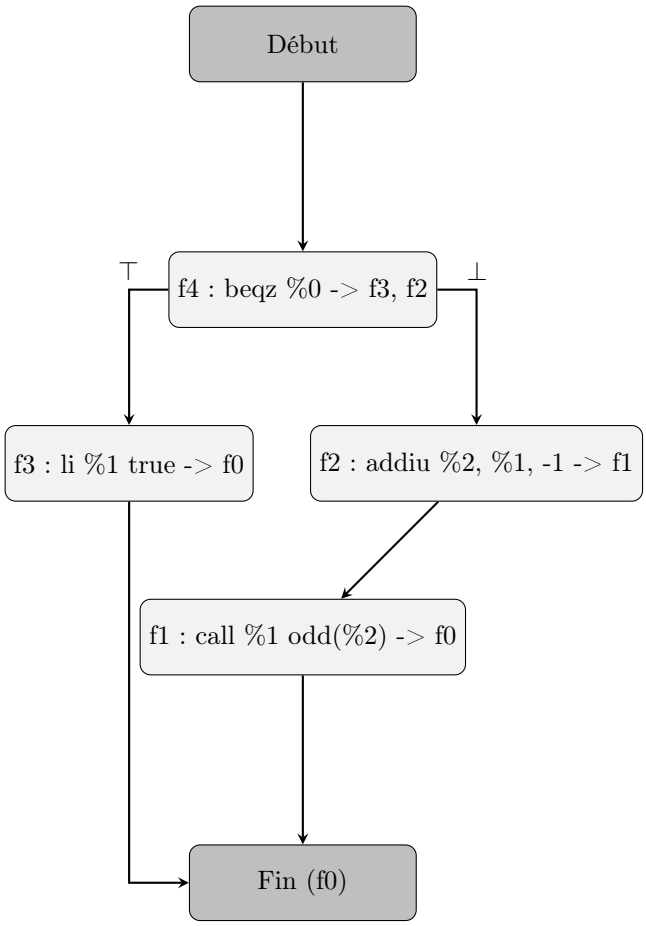
```
function even(n)
    if n = 0 then
        even := true
    else
        even := odd(n-1)

function odd(n)
```

```
if n = 0 then
    odd := false
else
    odd := even(n-1)
```

3. Pour procéder à la traduction en RTL, on alloue des pseudos registres aux paramètres, au retour ainsi qu'à l'ensemble des valeurs que l'on calcule. Ainsi :

$$(n, \%0), (retour, \%1), (n = 0, \%2), (n - 1, \%3)$$

Par exemple, pour *even*, on obtient :



qui se converti donc en :

```
function even(%0) %1
var %0 %1 %2
entry f4
exit f0
f4: blez %0 -> f3, f2
f3: li %1 true -> f0
f2: addiu %2, %0, -1 -> f1
f1: call %1 odd(%2) -> f0
```

```
function odd(%0) %1
var %0 %1 %2
entry f4
```

```
exit f0
f4: blez %0 -> f3, f2
f3: li %1 false -> f0
f2: addiu %2, %0, -1 -> f1
f1: call %1 even(%2) -> f0
```

4.

```
procedure even(1)
var %0 %1 %2 %3 %4 %5
entry f0
exit f18

f0: newframe -> f1
f1: move %3 $s0 -> f2
f2: move %4 $s1 -> f3
f3: move %5 $ra -> f4
f4: move %0 $a0 -> f5
f5: blez %0 -> f6, f7

f6: li %1, 1 -> f10

f7: addiu %2, %0, -1 -> f8
f8: move $a0, %2 -> f9
f9: call odd(1) -> f10
f10: move %1 $v0 -> f11

f11: move $s0 %3 -> f12
f12: move $s1 %4 -> f13
f13: move $ra %5 -> f14
f14: move $a0 %0 -> f15
f15: move $v0 %1 -> f16
f16: delframe -> f17
f17: jr $ra -> f18
```

```
procedure odd(1)
var %0 %1 %2 %3 %4 %5
entry f0
exit f18

f0: newframe -> f1
f1: move %3 $s0 -> f2
f2: move %4 $s1 -> f3
f3: move %5 $ra -> f4
f4: move %0 $a0 -> f5
f5: blez %0 -> f6, f7

f6: li %1, 1 -> f10

f7: addiu %2, %0, -1 -> f8
f8: move $a0, %2 -> f9
f9: call even(1) -> f10
f10: move %1 $v0 -> f11

f11: move $s0 %3 -> f12
```

```
f12: move $s1 %4 -> f13
f13: move $ra %5 -> f14
f14: move $a0 %0 -> f15
f15: move $v0 %1 -> f16
f16: delframe -> f17
f17: jr $ra -> f18
```

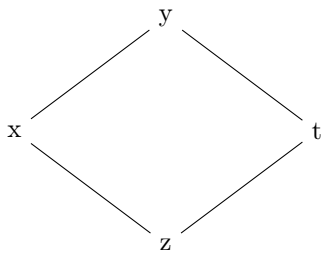
Oui, il est judicieux de les sauvegarde afin qu'il soit libre lors de l'attribution des registres aux variables

Exercice 2 (6 points) :

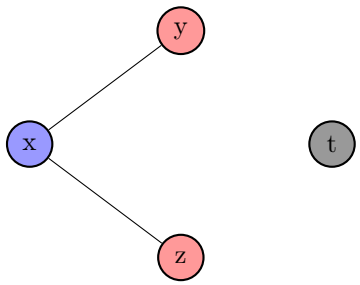
1. Donnez un exemple de programme PP qui permet d'avoir les interférences suivantes :
   — $(x, y), (x, z), (y, t), (z, t)$
   Justifier sa réponse en dessinant le graphe de flot de contrôle du programme, en faisant l'analyse de la durée de vie des variables (bien donner l'ensemble des variables vivantes à la fin du programme) et en dessinant le graphe d'interférences.
2. Colorier le graphe d'interférences obtenu à la question précédente avec 2 couleurs en utilisant l'algorithme de Chaitin . Doit-on "spiller" ?
   Même question en utilisant le "coalescing" (on attribute une couleur s'il y en a une disponible même si le sommet a été "spillé" dans un premier temps).

1. Soit le programme suivant :

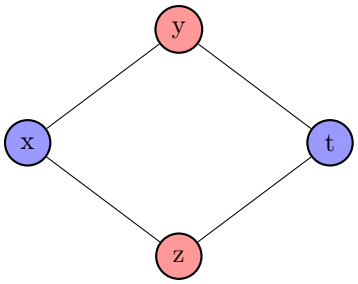
```
t = 2 {y, z}
x = t {t, y, z}
{y, z}
```



2. Colorions ce graphe avec 2 couleurs. Il n'y a pas de sommets de degré strictement inférieur à 2, on doit donc "spiller" t :



Si on utilise le coalescing, on remarque que z est coloriable.

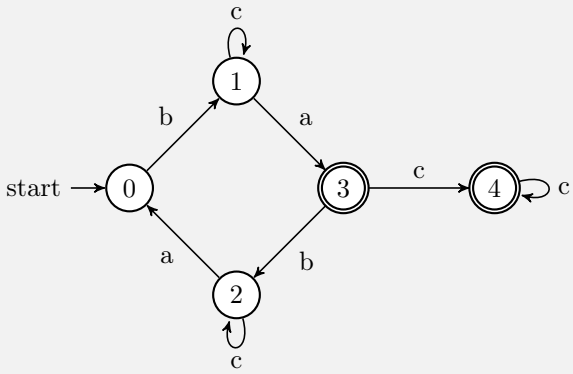

y
x
t
z

Exercice 3 (12 points) :
Nous disposons d'une machine virtuelle (VM) à registres proche de celle du cours mais très simplifiée. L'objectif est de générer un code de cette VM permettant l'interprétation d'automates déterministes, c'est-à-dire la reconnaissance d'un mot par un automate. On suppose que la VM peut gérer des listes LISP, avec des opérations spécialisées :

- **(car R1 R2)** prend la cellule dont l'adresse est dans le registre R1 et charge son champ *car* dans le registre R2 ; (**cdr** a le rôle symétrique pour le champ *cdr*).
- L'opération de comparaison **(cmp R1)**, utilisée avec un seul opérande, permet d'effectuer des tests de cellules (*consp, atom, null* en LISP) sur le contenu du registre R1 en positionnant les drapeaux de manière usuelle.
- **(bconsp #label)** est une instruction de branchement conditionnel qui effectue le branchement si le drapeau préalablement positionné (**cmp**) indique qu'il s'agit bien d'une cellule. Avec **batom** et **bnull** le branchement est conditionné au fait que la valeur testée est un atome ou *nil*.

Les conventions pour le code d'interprétation des automates sont les suivantes. La donnée (mot à reconnaître) est une liste de caractères (symboles), par exemple *(c b a c)* contenu dans le registre R0. A l'issue de l'exécution, la VM s'arrête et R0 contient l'état final atteint lors de l'exécution de l'automate, ou *nil*, suivant que le mot ait été reconnu ou pas.

Question 1 :
Soit l'automate ci-dessous, dont l'état initiale est 0 et les états finaux sont 3 et 4 :



1. Commencer par indiquer comment il est possible de traduire les états d'un automate dans le jeu d'instructions de la VM.
2. Ecrire le code VM correspondant à l'automate donné en exemple ci-dessus, en le commentant.

On suppose que l'on dispose, en LISP, d'un type de données abstrait automate, muni de l'interface fonctionnelle suivante :

- **(auto-etat-liste auto)** retourne la liste des états (entiers) de l'automate : pour celui de l'exemple. (0 1 2 3 4) ;
- **(auto-init auto)** retourne l'état (entier) initial de l'automate (0 dans l'exemple) ;
- **(auto-final-p auto etat)** retourne vrai si l'état argument est final (dans l'exemple, vrai pour 3 et 4, faux pour les autres)
- **(auto-trans-list auto etat)** qui retourne la liste des transitions issues de l'état argument, sous la forme d'une liste.

Question 2 :
Ecrire la fonction LISP auto2vm qui prend en argument un automate détermine (au sens de la structure de donnée précédente) et retourne le code VM correspondant (c'est-à-dire un code similaire à celui que vous avez écrit dans la question précédente pour l'automate donné en exemple).

- Spécifier le principe de la génération : comment traduire les états, les transitions, les états finaux, l'état initial, etc...
- Décomposer le problème en définissant des fonctions annexes pour traiter séparément chaque transition, chaque état, etc.

1. (a) 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

(b)
```
(vm-load '(
    (LABEL start) ; Debut de l'automate
    (JMP etat0)

    (LABEL etat0)
    (MOVE (:CONST etat0) R2)
    (CAR R0 R1)
    (CDR R0 R0)
    (CMP R1)
    (BNULL refuser)
    (CMP R1 (:CONST b))
    (JEQ etat1)
    (JMP refuser)

    (LABEL etat1)
    (MOVE (:CONST etat1) R2)
    (CAR R0 R1)
    (CDR R0 R0)
    (CMP R1)
    (BNULL refuser)
    (CMP R1 (:CONST c)
    (JEQ etat1)
    (CMP R1 (:CONST a)
    (JEQ etat3)
    (JMP refuser)

    (LABEL etat3)
    (MOVE (:CONST etat3) R2)
    (CAR R0 R1)
    (CDR R0 R0)
    (CMP R1)
    (BNULL accepter)
    (CMP R1 (:CONST c)
    (JEQ etat4)
    (CMP R1 (:CONST b)
    (JEQ etat2)
    (JMP refuser)

    (LABEL etat4)
    (MOVE (:CONST etat4) R2)
    (CAR R0 R1)
    (CDR R0 R0)
    (CMP R1)
    (BNULL accepter)
    (CMP R1 (:CONST c)
    (JEQ etat4)
    (JMP refuser)

    (LABEL etat2)
    (MOVE (:CONST etat2) R2)
```

```
    (CAR R0 R1)
    (CDR R0 R0)
    (CMP R1)
    (BNULL refuser)
    (CMP R1 (:CONST c)
    (JEQ etat2)
    (CMP R1 (:CONST a)
    (JEQ etat1)
    (JMP refuser)

    (LABEL refuser)
    (MOVE (:CONST nil) R0)
    (HALT)

    (LABEL accepter)
    (MOVE R2 R0)
    (HALT)
))
```

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 :

```
(defun gen-auto-transition (symbole-transition etat-suivant)
  (list
    `(CMP R1 '(:CONST ,symbole-transition)) ; Compare le symbole dans R1 avec le
        ↪ symbole de transition
    `(JEQ ,etat-suivant) ; Si les symboles correspondent, saute a l'etat suivant
  ))
```

Maintenant, il nous faut-être capable de convertir un état vers une liste d'instruction :

```
(defun gen-etat-automate (etat automate)
  (let ((transitions (auto-trans-list automate etat)))
    (append
      `((LABEL ,etat) ; Marque le debut de l'etat
        (CAR R0 R1) ; Charge le symbole actuel dans R1
        (CDR R0 R0) ; Avance dans la liste de symboles
        (CMP R1)) ; Compare le symbole actuel pour verifier s'il est nil
        (MOVE (:CONST ,etat) R2)
      (if (auto-final-p automate etat)
          `((BNULL accepter)) ; Si la liste est vide et l'etat est final, accepte le
              ↪  mot
          `((BNULL refuser))) ; Si la liste est vide mais l'etat n'est pas final,
              ↪ rejette le mot
      (mapcar (lambda (transition)
                (gen-auto-transition (first transition) (second transition)))
              transitions)
      '((JMP refuser)))))  ; Saute vers refuser si aucune transition n'est valide
```

Et enfin, pour finir, la génération de l'automate au complet :

```
(defun auto2vm (automate)
  (let ((etats (auto-etat-liste automate))
        (etat-initial (auto-init automate)))
    (append
      `((LABEL start) ; Debut de l'automate
```

```
      (JMP ,etat-initial)) ; Saute a l'etat initial
    (mapcar (lambda (etat) ; Genere les instructions pour chaque etat
              (gen-etat-automate etat automate))
            etats)
    `((LABEL accepter) ; Label pour acceptation
      (MOVE R2 R0) ; Indique une acceptation reussie
      (HALT) ; Fin de l'execution

      (LABEL refuser) ; Label pour refus
      (MOVE (:CONST nil) R0) ; Indique un echec
      (HALT))))) ; Fin de l'execution
```

2 Examen de 2021, session 1

Exercice 1 :
Soit la fonction f suivante :

$$f(n) = \begin{cases} n - 10, & \text{si } n > 100 \\ f(f(n + 11)), & \text{sinon} \end{cases}$$

1. Que calcule cette fonction pour $n \leq 101$? Justifier en déroulant son exécution sur au moins deux exemples.
2. Écrire la fonction f en PP.
3. Traduire la fonction f en UPP.
4. Traduire la fonction f en RTL.
5. Traduire la fonction f en ERTL.

1. Calculons des valeurs au hasard :

$$f(101) = 91$$
$$f(100) = f(f(111)) = f(101) = 91$$

   Il semblerait donc que cette fonction calcule toujours 91.

2.
```
function f(n: integer): integer
    if n > 100 then
        f := n - 10
    else
        f := f(f(n + 11));
```

3. Soit donc en UPP :

```
function f(n)
    if n > 100 then
        f := n - 10
    else
        f := f(f(n + 11));
```

4. Soit la traduction en RTL :

```
function f(%0): %1
var %0 %1 %2 %3 %4
entry f6
```

```
exit f0

f6: addiu %2 %0 -100 -> f5
f5: bgtz %2 -> f4, f3

f4: addiu %1 %0 -10 -> f0

f3: addiu %3 %0 11 -> f2
f2: call %4 f(%3) -> f1
f1: call %1 f(%4) -> f0
```

5. Et enfin, en ERTL :

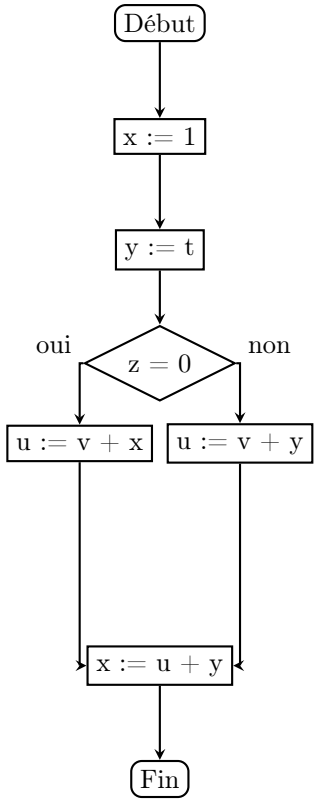
```
procedure f(1)
var %0 %1 %2 %3
f0: newframe -> f1
f1: move %0 $a0 -> f2
f2: move %1 $s0 -> f3
f3: move %2 $s1 -> f4
f4: move %3 $ra -> f5

f5: li $s0 100 -> f6
f6: slt $s1 $a0 $s0 -> f7
f7: bgtz $s1 -> f8, f9

f8: addiu $v0 $a0 -10 -> f14

f9: addiu $s0 $a0 11 -> f10
f10: move $a0 $s0 -> f11
f11: jal f -> f12
f12: move $a0 $v0 -> f13
f13: jal f -> f14

f14: move $a0 %0 -> f15
f15: move $s0 %1 -> f16
f16: move $s1 %2 -> f17
f17: move $ra %3 -> f18
f18: delframe -> f19
f19: jr $ra
exit f19
```

Exercice 2 :
Soit le programme PP suivant :

```
x := 1;
y := t;
if z = 0 then
    u := v + x
else
    u := v + y;
x := u + y
```

1. Dessiner le graphe de flot de contrôle de ce programme.
2. Faire une analyse de durée des variables sachant qu'à la fin du programme, x et y sont vivantes.
3. Dessiner le graphe d'interférences correspondant.
4. Colorier le graphe d'interférences avec 3 couleurs. Doit-on « spiller » x ? Mêmes questions avec 2 couleurs.

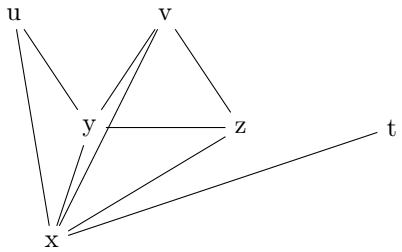
1.



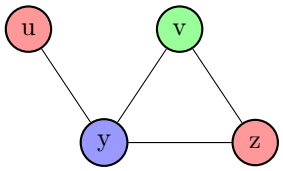
2. Analysons les durées de vie des variables :

```
x := 1; {v, t}
y := t; {v, x, t}
if z = 0 then {v, x, y, z}
    u := v + x {v, x, y}
else
    u := v + y; {v, y}
x := u + y {u, y}
{x, y}
```

3. Soit les interférences : $i = \{(x,y),(u,y),(u,x),(z,x),(z,y),(z,v),(y,v),(y,x),(x,v),(x,t)\}$



4. On ne peut pas colorier le graphe avec 3 couleurs sans spiller (le sommet x possède 4 voisins). Nous allons donc spiller x. Cela nous permet d'aussi éliminer t de la coloration car il devient un sommet isolé. On peut lui assigner le registre que l'on souhaite.



5. Si ici on veut procéder à une 2-coloration, il nous refaut spiller. Par exemple, nous allons spiller v