Compilation et interprétation (HMIN104)

Master AIGLE
Département Informatique
Faculté des Sciences de Montpellier

Examen du 15 janvier 2018

Tous les documents sont autorisés. Les ordinateurs portables sont également autorisés, mais sans le réseau.
L'examen dure 2h. Le barème est donné à titre indicatif. Le sujet comporte 4 pages et il y a 3 exercices.

Exercice 1 (7 pts)

Soit la fonction de Fibonacci :

$$f(n) = \begin{cases} 0, \text{ si } n = 0 \\ 1, \text{ si } n = 1 \\ f(n-1) + f(n-2), \text{ sinon} \end{cases}$$

1. Écrire la fonction f en PP.
   Traduire la fonction f en UPP.
2. Traduire la fonction f en RTL.
3. Traduire la fonction f en ERTL.
4. Traduire la fonction f en LTL (l'allocation de registres sera faite à la main sans passer par l'analyse de durée de vie des variables, le graphe d'interférences, et le coloriage correspondant).
   Traduire la fonction f en LIN.
5. Traduire la fonction f en MIPS.
6. Donner une version récursive terminale de f en PP.
   Traduire directement cette fonction en MIPS en optimisant au maximum l'utilisation de la pile (en minimisant le nombre d'allocations et de désallocations de trames de piles).

Ex 1 :

$$f(n) = \begin{cases} 0 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ f(n-1) + f(n-2), \text{ sinon} \end{cases}$$

1, **PP**

```
function f (n : integer ) : integer;
begin
    if  f <= 1 then f:= n
    else  f:= f(n-1) + f(n-2)
end;
```

UPP

```
function f(n);
begin
    f := 0;
    if  n <= 1 then f := n
    else f := f((-1+)n) + f((-2+)n)
end;
```

2, **RTL**

```
function f(%0) : %1
  var %0, %1, %2, %3, %4, %5, %6
  entry f1
  exit f0
f1: li %1, 0 → f2
f2: blez %0 → f3, f4        // if n = 0
f3: li %1, 0 → f0
f4: li %2, 1 → f5           // %2 = 1
f5: beq %0, %2 → f6, f7     // if n = 1
f6: li %1, 1 → f0
f7: addiu %3, %0, -1 → f8
f8: call %4, f(%3) → f9
f9: addiu %5, %0, -2 → f10
f10: call %6, f(%5) → f11
f11: add %1, %4, %6 → f0
f21: move $s1, %7 → f22
f22: delframe → f23
f23: jr $ra
```

ERTL

```
procedure f(1)
  var %0, %1, %2, %3, %4, %5, %6, %7
  entry f1
f1 : newframe → f2
f2 : move %5 , $ra → f3
f3 : move %6 , $s0 → f4
f4 : move %7 , $s1 → f5
f5 : move %0 , $a0 → f6   // $a0 là n
f6 : li %1, 0 → f7
f7 : slt %2, %0, 2 → f8
f8 : beq %2, 1 → f9 , f10
f9 : li %1, %0 → f0
f10 : addiu %3, %0, -1 → f11
f11 : move $a0, %3 → f12
f12 : call f(1) → f13
f13 : move $s0, $v0 → f14   // f(n-1)
f14 : addiu %4 , %0, -2 → f15
f15 : move $a0 , %4 → f16
f16 : call f(1) → f17
f17 : move $s1 , $v0 → f18   // f(n-2)
f18 : addiu %1, $s0 , $s1 → f0
f0 : j → f19
f19 : move $ra , %5 → f20
f20 : move $s0, %6 → f21
```

Exercice 2 (5 pts)

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 » ? Mêmes questions avec 2 couleurs.

Exercice 3 - Compilation d'automates vers une VM (12 pts)

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 :

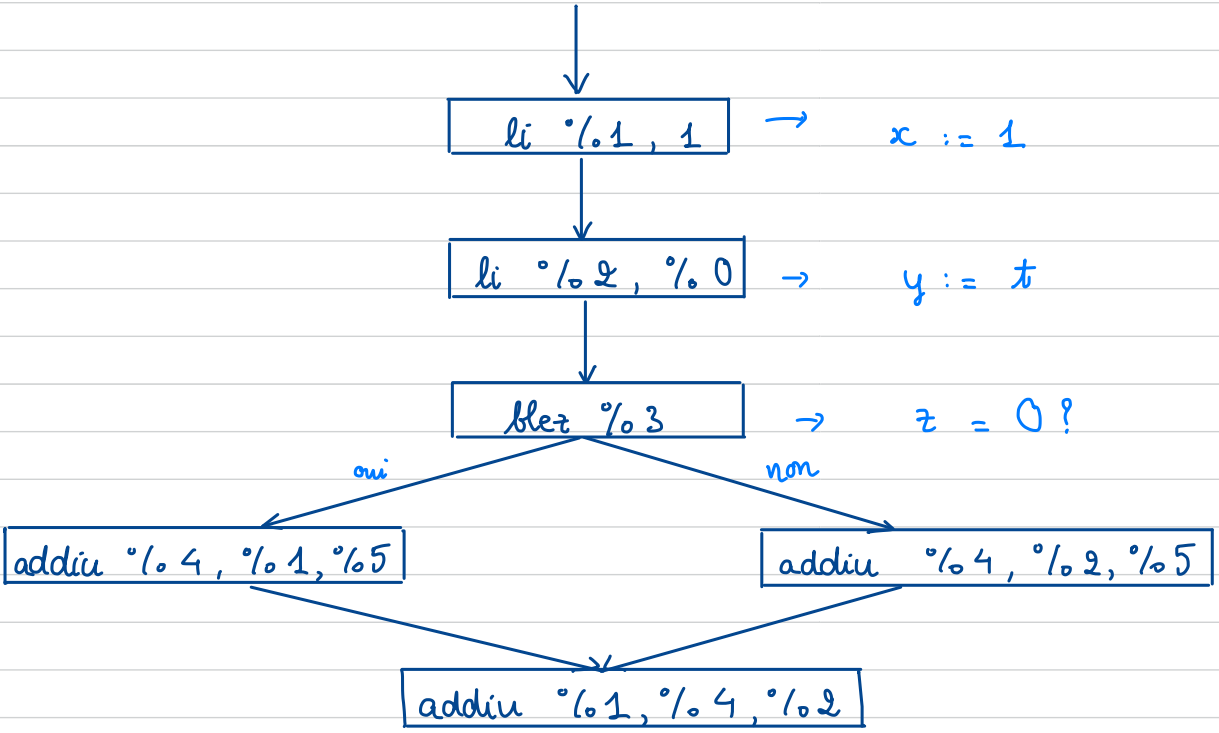
— `(car R1 R2)` prend la cellule dont l'adresse est dans le registre `R1` et charge son champ `car` dans le registre `R2` ; (`cdr` a le rôle symétrique pour le champ `cdr`) ;

— 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 des drapeaux de manière usuelle ;

— `(bconsp #label)` est une instruction de branchement conditionnel qui effectue le branchement si le drapeau préalablement positionné par `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`.

Ex2 :

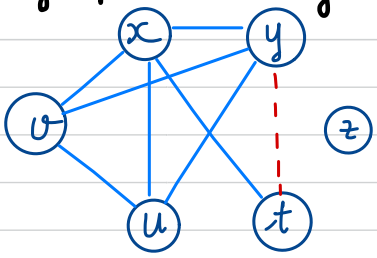
1, Graphe de flot de contrôle de ce programme



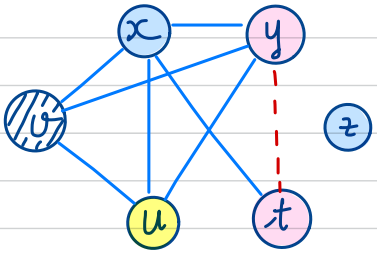
2, Analysé de duréé

{ v, t }
x := 1
{ v, x, t }
y := t
{ v, x, y }
z := 0
{ v, x, y }
u := v + x
{ v, x, y }
u := v + y
{ u, y }
x := u + y
{ x, y }

3, Graphe interférence

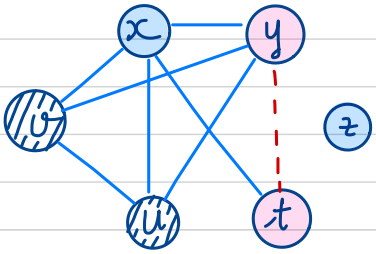


4, Colorié 3 couleurs



On spille (v)

Colorié 2 couleurs



On spille (v) (u)

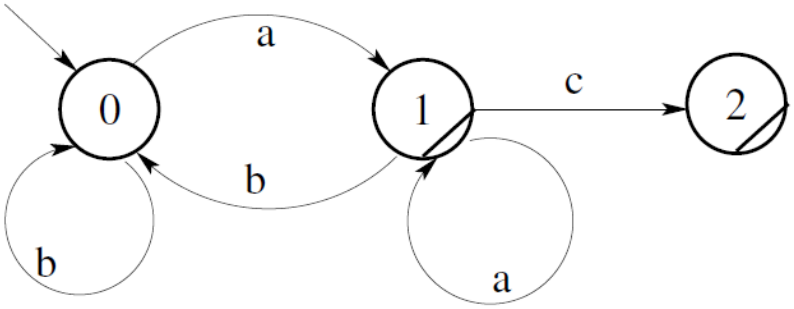
Les conventions pour le code d'interprétation des automates sont les suivantes. La donnée (mot à reconnaître) est une liste de caractères (symboles), par exemple `(b b a a a)`, contenue dans le registre `R0`. A l'issue de l'exécution, la VM s'arrête et `R0` contient l'état final atteint lors de l'exécution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

Question 1

Pour éviter les ambiguités, commencez par définir précisément et succinctement les instructions de la VM dont vous vous servez. (NB : ne décrivez pas les instructions inutilisées de la VM.)

Question 2

Soit l'automate déterministe suivant, dont l'état initial est 0 et les états finaux sont 1 et 2 :



1. Commencez par indiquer comment il est possible de traduire les états de l'automate dans la VM. Une pile est-elle nécessaire ? Pourquoi ?
2. Écrire 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 la figure, (0 1 2)
— `(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 1 et 2, faux pour 0) ;
— `(auto-trans-list auto etat)` retourne la liste des transitions issues de l'état argument, sous la forme d'une liste.

Question 3

Écrire la fonction LISP `auto2vm` qui prend en argument un automate déterministe (au sens de la structure de données précédente) et retourne le code VM correspondant (c'est-à-dire un code voisin de celui que vous avez écrit dans la question précédente pour l'automate de la figure).

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

Question 4 (bonus)

Indiquer dans les grandes lignes comment il faudrait généraliser cela à des automates non-déterministes ? Rappelons qu'un automate est déterministe si, de chaque état, chaque caractère de l'alphabet considéré conduit à au plus une transition. C'est le cas de celui de l'exemple.

* * * * * * * * * * * * *

HMIN104 - Correction Exam 2018

27 novembre 2019

1 Partie Delahaye

1.1 Pseudo Pascal

```
f(n : integer) : integer
    if n <= 1 then
        f := n
    else
        f:= f(n - 1) + f(n - 2)
```

1.2 UPP

```
f(n)
    if n <= 1 then
        f := n
    else
        f:= f(n - 1) + f(n - 2)
```

1.3 RTL

```
function f(%0): %1
var %0, %1, %2, %3, %4, %5, %6
entry f1
exit f0

f1: li %1, 0          -> f2
f2: slt %2, %0, 2     -> f3
f3: beq %2, 1         -> f5, f4
f4: addiu %3, %0, -1 -> f6
f6: call %4, f(%3)    -> f7
f7: addiu %5 %0, -2   -> f8
f8: call %6, f(%5)    -> f9
f9: addiu %1, %4, %6 -> f0
```

→ slt set on less then
↳ bool → 1 si less
→ 0 si bigger

```
f5: li %1, %0        -> f0
```

1.4 ERTL

```
procedure f(1)
var %0, %1, %2, %3, %4, %5, %6, %7, %8, %9
entry f1

f1 : newframe            -> f2
f2 : move %7, $ra        -> f3
f3 : move %8, $s0        -> f4
f4 : move %9, $s1        -> f22
f22: move %0, $a0        -> f5
f5 : li %1, 0            -> f6
f6 : slt %2, %0, 2       -> f7
f7 : blez %2             -> f8, f22
f8 : addiu %3, %0, -1    -> f9
f9 : move $a0, %3        -> f10
f10: call(f1)            -> f11
f11: move $s0, $v0       -> f12
f12: addiu %5, %0, -2    -> f13
f13: move $a0, %5        -> f14
f14: call f(1)           -> f15
f15: move $s1, $v0       -> f16
f16: addiu %1, $s1, $s0  -> f0
f17: move $ra, %7        -> f18
f18: move $s0, %8        -> f19
f19: move $s1, %9        -> f20
f20: delframe            -> f21
f21: jr $ra              -> f22
f22: li %1, %0           -> f0
```

1.5 LTL

```
procedure f(1)
var 11
entry f1

f1 : newframe            -> f2
f2 : sets local(0), $ra  -> f3
f3 : sets local(4), $s0  -> f4
f4 : move $s0, $a0       -> f5
f5 : slt $a0, $a0, 2     -> f6
f6 : bleq $a0, 1         -> f7, f8
f8 : addiu $a0, $s0, -1  -> f9
```

```
f9 : call f              -> f11
f11: sets local(8) $s1   -> f12
f12: move $s1, $v0       -> f13
f13: addiu $a0, $s0, -2  -> f14
f14: call f              -> f15
f15: move $s0, $v0       -> f16
f16: addiu $v0, $s1, $s0 -> f0
f0 : j                   -> f17
f17: gets $ra, local(0)  -> f18
f18: gets $s0, local(4)  -> f19
f19: gets $s1, local(8)  -> f20
f20: delframe            -> f21
f21: jr $ra              -> f22
f7 : move $v0, $a0         -> f0
```

1.6 LIN

```
procedure f(1)
var 11
f1 :
newframe
sets local(0), $ra
sets local(4), $s0
move $s0, $a0
slt $a0, $a0, 2
bleq $a0, 1, f7
addiu $a0, $s0, -1
call f
sets local(8) $s1
move $s1, $v0
addiu $a0, $s0, -2
call f
move $s0, $v0
addiu $v0, $s1, $s0
f17:
gets $ra, local(0)
gets $s0, local(4)
gets $s1, local(8)
delframe
jr $ra
f7 :
move $v0, $a0
j f17
```

1.7 MIPS

```
fib:
    subi $sp $sp, 12
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    move $s0, $a0
    slt $a0, $s0, 2
    beq $ao, 1, fib1

    sw $s1, 8($sp)
    move $s1, $v0
    addiu $a0, $s0, -2
    jal fib
    addiu $v0, $s1, $v0
    f17 :
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    addiu $sp, $sp, 12
    jr $ra
    fib1:
    move $v0, $a0
    j f17
```

1.8 Version récursive terminale en PP

```
f(n: integer, acc1 : integer, acc2 : integer) : integer
    if n <= 1 then
        f := n
    else
        f := f(n - 1, acc2, acc1 + acc2)

f(n, 0, 1)
```

1.9 Version récursive terminale en MIPS

```
addiu $a0, $a0, -1
move $a1, $a2
move $a2, $a2, $a1
lw $ra, 0($sp)
addiu $sp, 4
j fib
```

2 Partie Lafourcade

Question 1

- HALT → quit
- LABEL → đặt tên cho état
- CMP → so sánh
- MOVE
- JEQ → jump if equal

Question 2.1

Si on associe un label (une étiquette) à chaque état alors ces derniers seront liés à une adresse. Ainsi en utilisant l'instruction JMP on pourrait facilement passer d'un état à un autre. Ainsi une pile n'est pas nécessaire.

Question 2.2

```
(LABEL E0)
(CAR R0 R1)
(CDR R0 R0)
(CMP R1 a)
(JEQ E1)
(CMP R1 b)
(JEQ E0)
(MOVE NIL R0)
(HALT)

(LABEL E1)
(MOVE 'E1 R2)
(cmp R0)
(bnull H)
(CAR R0 R1)
(CDR R0 R0)
(CMP R1 a)
(JEQ E1)
(CMP R1 b)
(JEQ E0)
(CMP R1 c)
(JEQ E2)
(MOVE NIL R0)
(HALT)

(LABEL E2)
(MOVE 'E2 R2)
(cmp R0)
(bnull H)
```

điểm chứa
move từ trái qua phải
liste
if bnull → Label H
car(R0) cho vào R1
return
batom
or
bnull
jump if equal

```
(CAR R0 R1)
(CDR R0 R0)
(MOVE NIL R0)
(HALT)

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

Question 3