

Schriftelijk Examen: “Interpretatie van Computerprogramma’s I”

Eerste zitting, academiejaar 2017–2018
Coen De Roover
Vrije Universiteit Brussel

25 juni 2018

Het examen bestaat uit 5 vragen, verdeeld over 7 bladzijden.
Gelieve elke vraag te beantwoorden op een afzonderlijk blad.
Zet je naam, studierichting en rolnummer op *elk* blad dat je afgeeft.

De vragen zijn geformuleerd ten opzichte van de gepubliceerde slides en het referentieboek.
Deze mogen, samen met *persoonlijke* nota's, geraadpleegd worden tijdens deze test. Het gebruik van elektronische media is echter *niet* toegestaan.
Veel succes!

1 Meta-circulaire evaluator

Breid de meta-circulaire evaluator uit met ondersteuning voor zogenaamde `let-match` expressies. De syntax van deze expressies is als volgt:

```
1 (let-match ((<var_1> ... <var_n>) <l_exp>)  
2   <b_exp_1> ... <b_exp_m>)
```

Als `<l_exp>` evalueert naar een lijst van evenveel elementen als de lijst van variabelen `(<var_1> ... <var_n>)`, bindt `let-match` elk element uit de lijst waarnaartoe `<l_exp>` evalueerde aan de overeenkomstige variabele uit `(<var_1> ... <var_n>)` tijdens de evaluatie van `<b_exp_1> ... <b_exp_m>`.

Volgend transcript licht het gebruik van `let-match` verder toe:

```

1  ;;; M-Eval input:
2  (let-match ((x y) '(1 2))
3    (+ x y))
4  ;;; M-Eval value:
5  3
6
7  ;;; M-Eval input:
8  (let-match ((a b c d) (map sqrt '(1 4 9 25)))
9    ((lambda (x) x) b)
10   (+ a b c d))
11  ;;; M-Eval value:
12  11
13
14  ;;; M-Eval input:
15  (let-match ((i j) '(1))
16    i)
17  **ERROR: let-match lists of different lengths**

```

- 1.a) Implementeer alle hulpprocedures die de meta-circulaire evaluator nodig heeft om let-match expressies te herkennen en er onderdelen uit te selecteren.
- 1.b) Breid de meta-circulaire evaluator uit met ondersteuning voor let-match expressies. Implementeer let-match als afgeleide expressie (bv. aan de hand van een procedure let-match->exp).
- 1.c) Breid de meta-circulaire evaluator opnieuw uit met ondersteuning voor let-match expressies. Implementeer let-match ditmaal als een “special form” met een “dedicated evaluator” procedure (bv. eval-let-match).

2 Logisch Programmeren

Beschouw volgende feitenbank over bankrekeningen (accounts) en overschrijvingen (transfers):

```

1  ;(account <rekeningnummer> <naam> <vertrouwde accounts>)
2  (account "BE-AA-0123" "Coen" ("BE-AA-0246" "BE-AA-0369"))
3  (account "BE-AA-0246" "Thierry" ("BE-AA-0123" "BE-AA-0369"))
4  (account "BE-AA-0369" "Ward" ("BE-AB-4589" "BE-AA-0123"))
5  (account "BE-AA-0482" "Wolfgang" ())
6
7  ;(transfer <id> <bron> <bestemming> <tijdstip> <EUR>)
8  (transfer 0 "BE-AA-0123" "BE-AA-0246" "2018-06-25 10:00" 15)
9  (transfer 1 "BE-XX-5672" "BE-AA-0246" "2018-06-25 11:20" 500)
10 (transfer 2 "BE-AA-0246" "BE-XY-2382" "2018-06-25 11:25" 500)
11 (transfer 3 "BE-AA-0246" "BE-YX-3962" "2018-06-25 11:26" 250)

```

Van bankrekeningen is het rekeningnummer gekend, de naam van de eigenaar, en een lijst van vertrouwde accounts. Van overschrijvingen is een volgnummer gekend, de rekeningnummers van de bron en de bestemming, het tijdstip, en het bedrag in euro.

- 2.a) Geef een query die alle accounts selecteert waarvoor geen vertrouwde account gekend is. Bij bovenstaande feitenbank zou dit dus de account van Wolfgang teruggeven.
- 2.b) Geef een regel met hoofd (wordt-vertrouwd-door ?vertrouwde ?account) die slaagt wanneer ?vertrouwde als vertrouwd geregistreerd staat bij de account met rekeningnummer ?account.

De regel moet onder andere volgende interactie toelaten:

```
1   ;;; Query input:
2   (wordt-vertrouwd-door "BE-AA-0123" ?account)
3   ;;; Query results:
4   (wordt-vertrouwd-door BE-AA-0123 BE-AA-0369)
5   (wordt-vertrouwd-door BE-AA-0123 BE-AA-0246)
```

2.c) Geef een regel met hoofd (doorgifte ?in ?uit ?bedrag) die slaagt wanneer aan volgende voorwaarden voldaan is:

- ?in en ?uit zijn de identifiers van transacties met waarde ?bedrag
- de bestemming van transactie ?in is gelijk aan de bron van transactie ?uit
- de transactie ?uit vindt plaats na transactie ?in. Je mag er hiervoor van uitgaan dat een Scheme functie is-later-dan bestaat waarmee je twee data kan vergelijken.

Voor bovenstaande feitenbank laat de regel volgende interactie toe:

```
1   ;;; Query input:
2   (doorgifte ?id-in ?id-uit ?bedrag)
3   ;;; Query results:
4   (doorgifte 1 2 500)
```

2.d) Geef een regel met hoofd (verdachte-doorgifte ?in ?uit) die slaagt wanneer ?in en ?uit transacties identificeren die betrokken zijn bij een doorgifte (zie vraag 2.c) met een bedrag hoger dan €500, en waarbij ?uit geen transactie naar een vertrouwde rekening was. Voor bovenstaande feitenbank laat dat de volgende interactie toe:

```
1   ;;; Query input:
2   (verdachte-doorgifte ?id-in ?id-uit)
3   ;;; Query results:
4   (verdachte-doorgifte 1 2)
```

2.e) Geef een regel met hoofd (vertrouwd-namen ?account ?namen) die slaagt wanneer ?namen een lijst is met de namen van de eigenaars van vertrouwde accounts van account ?account. Deze regel laat onder andere volgende interactie toe:

```
1   ;;; Query input:
2   (vertrouwd-namen "BE-AA-0123" ?namen)
3   ;;; Query results:
4   (vertrouwd-namen BE-AA-0123 (Thierry Ward))
```

3 Automatisch geheugenbeheer

Beschouw de Scheme versie van het stop-and-copy garbage collection algoritme:

```
1 (define (RECLAIM)
2   (define (relocate-old-result-in-new val)
3     (if (pair? val)
4       (let*
5         ((old (VALUE val))
6          (old-car (vector-ref THE-CARS old))
7          (old-cdr (vector-ref THE-CDRS old)))
8         (if (MASK? old-car)
9             old-cdr
10            (let*
11              ((new FREE)
12               (pair (MAKE PAIR new)))
13              (set! FREE (+ FREE 1))
14              (vector-set! NEW-CARS new old-car)
15              (vector-set! NEW-CDRS new old-cdr)
16              (vector-set! THE-CARS old MASK)
17              (vector-set! THE-CDRS old pair)
18              pair)))
19     val))
20 (define (gc-flip) ...)
21 (define (gc-loop scan)
22   (if (= scan FREE)
23       (gc-flip)
24       (let
25         ((old-car (vector-ref NEW-CARS scan))
26          (old-cdr (vector-ref NEW-CDRS scan)))
27         (vector-set! NEW-CARS scan (relocate-old-result-in-new old-car))
28         (vector-set! NEW-CDRS scan (relocate-old-result-in-new old-cdr))
29         (gc-loop (+ scan 1))))))
30 (set! FREE 0)
31 (set! ROOT (relocate-old-result-in-new ROOT))
32 (gc-loop 0))
```

Deze implementatie van het algoritme voert in procedure `gc-loop` voor sommige adressen in het geheugen tot tweemaal toe overbodig werk uit.

- 3.a) Beschrijf het overbodige werk en de geheugenadressen waarvoor dit uitgevoerd wordt.
- 3.b) Geef een performantere versie van het algoritme door dit overbodige werk te elimineren.

4 Programmeren van registermachines

De driehoek van Pascal werd voor het eerst gebruikt door de Franse wiskundige Blaise Pascal in de 17e eeuw:

																1																
															1		1															
														1		2		1														
													1		3		3		1													
												1		4		6		4		1												
											1		5		10		10		5		1											
										1		6		15		20		15		6		1										
									1		7		21		35		35		21		7		1									
								1		8		28		56		70		56		28		8		1								
							1		9		36		84		126		126		84		36		9		1							
						1		10		45		120		210		252		210		120		45		10		1						
					1		11		55		165		330		462		462		330		165		55		11		1					
				1		12		66		220		495		792		924		792		495		220		66		12		1				
			1		13		78		286		715		1287		1716		1716		1287		715		286		78		13		1			
		1		14		91		364		1001		2002		3003		3432		3003		2002		1001		364		91		14		1		
	1		15		105		455		1365		3003		5005		6435		6435		5005		3003		1365		455		105		15		1	
1		16		120		560		1820		4368		8008		11440		12870		11440		8008		4368		1820		560		120		16		1

Om een getal in deze driehoek te construeren moet je de twee getallen die erboven staan optellen. Op de uiteinden wordt altijd 1 geschreven. De volgende Scheme functie berekent, gegeven een rij, de volgende rij in de driehoek:

```
1 (define (volgende-rij rij)
2   (define (rij-iter oude-rij nieuwe-rij)
3     (if (null? (cdr oude-rij))
4       (cons 1 nieuwe-rij)
5       (rij-iter (cdr oude-rij)
6                 (cons (+ (car oude-rij)
7                           (cadr oude-rij))
8                       nieuwe-rij))))
9   (rij-iter rij '(1)))
```

Het volgende transcript licht het gebruik van deze functie verder toe:

```
1 > (volgende-rij '(1))
2 (1 1)
3 > (volgende-rij '(1 3 3 1))
4 (1 4 6 4 1)
```

Zet de Scheme functie `volgende-rij` om naar een subroutine in de registermachinetaal. De registermachine waarop jouw code uitgevoerd zal worden, kent de volgende primitieve operaties: `list`, `car`, `cdr`, `null?`, `cons` en `+`. De registermachine beschikt verder over volgende registers: `v1`, `v2`, `cont` en `res`.

Je subroutine begint met een gepast label waar naartoe gesprongen kan worden. De input voor je subroutine zit in register `v1`. Je moet je resultaat uitschrijven in register `res`. Aan het einde moet je

subroutine terugspringen naar de originele inhoud van het cont register.

```
1 (assemble-and-run '(start
2   (assign v1 (op list) (const 1)
3     (const 3)
4     (const 3)
5     (const 1))
6   (assign cont (label stop))
7   (goto (label volgende-rij-start))
8
9   <JOUW CODE HIER>
10
11   stop))
```

5 Compilatie

Onderstaand codefragment werd gegenereerd door een oproep van de procedure `compile`. Re-construeer de *drie argumenten* van de oproep. Merk op dat we de labels geanonimiseerd hebben.

```
1      (assign t (op make-compiled-procedure) (label label-1) (reg env))
2      (goto (label label-11))
3  label-1
4      (assign env (op compiled-procedure-env) (reg proc))
5      (assign env (op extend-environment) (const (s)) (reg argl) (reg env))
6      (save continue)
7      (save env)
8      (assign proc (op lookup-variable-value) (const l) (reg env))
9      (assign val (op lookup-variable-value) (const s) (reg env))
10     (assign argl (op list) (reg val))
11     (assign val (const 7))
12     (assign argl (op cons) (reg val) (reg argl))
13     (test (op primitive-procedure?) (reg proc))
14     (branch (label label-3))
15  label-2
16     (assign continue (label label-4))
17     (assign val (op compiled-procedure-entry) (reg proc))
18     (goto (reg val))
19  label-3
20     (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
21  label-4
22     (restore env)
23     (restore continue)
24     (test (op false?) (reg val))
25     (branch (label label-9))
26  label-5
27     (assign proc (op lookup-variable-value) (const c) (reg env))
28     (assign val (op lookup-variable-value) (const r) (reg env))
29     (assign argl (op list) (reg val))
30     (test (op primitive-procedure?) (reg proc))
31     (branch (label label-7))
32  label-6
33     (assign val (op compiled-procedure-entry) (reg proc))
34     (goto (reg val))
35  label-7
36     (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
37     (goto (reg continue))
38  label-8
39  label-9
40     (assign val (op lookup-variable-value) (const w) (reg env))
41     (goto (reg continue))
42  label-10
43  label-11
```
