

Schriftelijk Examen:  
“Interpretatie van Computerprogramma’s I”

Tweede zitting, academiejaar 2016–2017  
Coen De Roover  
Vrije Universiteit Brussel

28 augustus 2017

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 *doto* expressies. De syntax van deze expressies is als volgt:

---

```
1 (doto <exp>
2   (<operator_1> <operand_11> ... <operand_1x>)
3   ...
4   (<operator_n> <operand_n1> ... <operand_ny>))
```

---

Zulk een *doto* neemt een expressie *<exp>* als argument gevolgd door een aantal applicatie-expressies (*<operator\_1> <operand\_11> ... <operand\_1x>*). Bij het evalueren van de *doto*, krijgt elk van deze applicatie-expressies het resultaat van de voorgaande expressie als extra, eerste operand.

Volgend transcript licht het gebruik van `doto` verder toe:

---

```
1  ;; M-Eval input:
2  (doto 5)
3  ;; M-Eval value:
4  5
5  ;; M-Eval input:
6  (doto 5 (+ 1))
7  ;; M-Eval value:
8  6
9  ;; M-Eval input:
10 (doto 5 (+ 1) (* 2) (= 12))
11 ;; M-Eval value:
12 #t
```

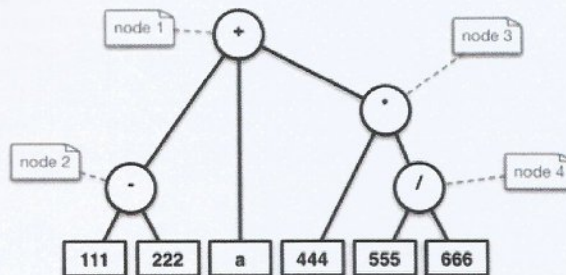
---

Merk op dat de `doto`-expressie zelf evalueert naar de waarde van de laatste applicatie-expressie (of de waarde van `<exp>` indien er geen procedure-applicaties opgegeven werden).

- 1.a) Implementeer alle hulpfuncties die je nodig hebt om de meta-circulaire evaluator uit te breiden met ondersteuning voor `doto`-expressies.
- 1.b) Breid de meta-circulaire evaluator uit met ondersteuning voor `doto`-expressies. Implementeer `doto` als afgeleide expressie (bv. aan de hand van een procedure `doto->application`).
- 1.c) Breid de meta-circulaire evaluator opnieuw uit met ondersteuning voor `doto`-expressies. Implementeer `cas` als een "special form" met een "dedicated evaluator" procedure (bv. `eval-doto`).

## 2 Logisch Programmeren

Rekenkundige expressies kunnen als een boom voorgesteld worden. De expressie  $(+ (- 111 222) a (* 444 (/ 555 666)))$  kan bijvoorbeeld voorgesteld worden als de volgende boom:



Equivalent kan deze expressie voorgesteld worden als een reeks knopen, die geïdentificeerd worden aan de hand van hun ID (bv. 1, 2, 3 en 4 in bovenstaande expressie). Knopen kunnen onmiddellijke waarden voorstellen (bv. 111, 222, a, etc. in bovenstaande expressie), maar ook verwijzingen naar andere knopen. Zo heeft knoop 1 verwijzingen naar knopen 2 en 3.

Als notatie gebruiken we VAL om onmiddellijke waarden neer te schrijven en REF voor verwijzingen, zodat bv. (VAL 9) het getal 9 voorstelt, (VAL x) de variabele x aanduidt en (REF 123) een verwijzing is naar de knoop met ID 123. We voegen rekenkundige expressies aan de databank van de logische query evaluator toe door ze als logische feiten van de vorm (node NODE-ID OPERATOR OPERANDS) in te geven. Voor bovenstaande expressie wordt dat dus:

---

```
1 (node 1 (VAL +) ((REF 2) (VAL a) (REF 3)))
2 (node 2 (VAL -) ((VAL 111) (VAL 222)))
3 (node 3 (VAL *) ((VAL 444) (REF 4)))
4 (node 4 (VAL /) ((VAL 555) (VAL 666)))
```

---

**2.a)** Implementeer het binaire predicaat `node-with-operator` aan de hand van een regel die de ID van alle nodes met een bepaalde operator zoekt, of voor een gegeven ID de operator bepaalt. Volgend transcript licht het predicaat toe:

---

```

1 > (node-with-operator + ?id)
2 ;;; Query results:
3 (node-with-operator + 1)
4 > (node-with-operator ?op 3)
5 ;;; Query results:
6 (node-with-operator * 3)
7 > (node-with-operator ?op ?id)
8 ;;; Query results:
9 (node-with-operator / 4)
10 (node-with-operator * 3)
11 (node-with-operator - 2)
12 (node-with-operator + 1)

```

---

- 2.b) Implementeer het binaire predicaat `node-with-any-operator-of` dat knopen selecteert waarvan de operator zich in de meegegeven lijst bevindt. Volgend transcript licht het predicaat toe:

---

```

1 > (node-with-any-operator-of (+ -) ?id)
2 ;;; Query results:
3 (node-with-any-operator-of (+ -) 1)
4 (node-with-any-operator-of (+ -) 2)

```

---

- 2.c) Implementeer het unaire predicaat `can-be-trivially-simplified` dat slaagt wanneer een knoop triviaal vereenvoudigd kan worden. Een knoop kan triviaal vereenvoudigd worden als alle operanden constante getallen zijn. Een operand is een constant getal als het een expressie van de vorm `(VAL ?n)` is, waarbij `?n` een getal is. Je kan de Scheme procedure `number?` gebruiken om dat laatste na te gaan.

Van de bovenstaande boom voldoen dus enkel de expressies `(- 111 222)` en `(/ 555 666)`, d.i., knopen 2 en 4:

---

```

1 > (can-be-trivially-simplified ?id)
2 ;;; Query results:
3 (can-be-trivially-simplified 2)
4 (can-be-trivially-simplified 4)

```

---

- 2.d) Bovenstaande regel was nodeloos strikt: ook knoop 3 hangt niet af van variabelen, en kan dus vereenvoudigd worden. Implementeer aan de hand van `can-be-trivially-simplified` het unaire predicaat `can-be-simplified` zodat de volgende interactie mogelijk wordt:

---

```
1 > (can-be-simplified ?id)
2 ;; Query results:
3 (can-be-simplified 2)
4 (can-be-simplified 3)
5 (can-be-simplified 4)
```

---

### 3 Automatisch geheugenbeheer

Beschouw de Scheme versie van het stop-and-copy garbage collection algoritme uit de slides. Deze gebruikt zogenaamde “broken hearts” en “forwarding addresses” als oplossing voor het probleem van meerdere verwijzingen naar eenzelfde `cons`-cel. Het doel is informatie te verzamelen over hoe vaak dit fenomeen zich voordoet.

- 3.a) Pas de implementatie aan zodanig dat voor elke cel bijgehouden wordt hoeveel verwijzingen ernaartoe bestaan. Dit echter zonder tellertjes bij te houden buiten de reeds bestaande vectoren `THE-CARS`, `THE-CDRS`, `NEW-CARS`, of `NEW-CDRS`. Bestudeer hiervoor de manier waarop broken hearts geïmplementeerd zijn als getypeerde pointers. Er is daar namelijk nog wat ruimte.
- 3.b) Breid je vorige aanpassing uit zodanig dat het maximale aantal verwijzingen naar eenzelfde cel geprint wordt aan het einde van elke garbage collection.

## 4 Programmeren van registermachines

Onderstaande procedure `prime-test` bepaalt met behulp van de procedure `sqrt` efficiënt of een getal priem is:

---

```
1 (define (sqrt n)
2   (define (sqrt-loop i)
3     (if (> (* i i) n)
4         i
5         (sqrt-loop (+ i 1))))
6   (sqrt-loop 0))
7
8 (define (prime-test n)
9   (define s (sqrt n))
10  (define (prime-test-loop i)
11    (if (= i s)
12        true
13        (if (= (% n i) 0)
14            false
15            (prime-test-loop (+ i 1)))))
16  (prime-test-loop 2))
```

---

Vertaal beide procedures naar registermachinetaal. Je mag ervan uitgaan dat je registermachine de benodigde rekenkundige operaties ondersteunt, en over de registers `cont`, `res`, `v1`, `v2` beschikt.

Gebruik volgend patroon voor een vertaling van de procedures en van de oproep (`prime-test 101`):

---

```
1 start
2   (assign v1 (const 101))
3   (assign cont (label stop))
4   (goto (label prime-start))
5
6 sqrt-start
7   ...
8
9 prime-test-start
10  ...
11
12 stop
```

---

Zorg ervoor dat, na uitvoering, het `res` register het resultaat bevat en dat de machine naar het door de oproeper gekozen label is gesprongen.

## 5 Compilatie

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

---

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

---