

TÖL304G

Forritunarmál

Vikublað 2

Snorri Agnarsson

24. ágúst 2024

Efnisyfirlit

1	Viðfangaflutningar	2
1.1	Gildisviðföng	2
1.2	Tilvísunarviðföng	2
1.3	Afritsviðföng	3
1.4	Löt viðföng og nafnviðföng	3
2	Dæmi	3
3	Scheme	5
4	Forritspróun í Scheme	5
4.1	Ef þið notið DrRacket	6
5	Endurkvæmni og halaendurkvæmni	6
6	Nokkur Scheme föll	6
6.1	Lélegt <i>reverse</i> -fall	7
6.2	Gott <i>reverse</i> -fall	8
6.3	Halaendurkvæmt Fibonacci-fall	8
6.4	„Venjulegt“ fall til að reikna $n!$	9
6.5	Halaendurkvæmt fall til að reikna $n!$	9
6.6	Einfalt map fall	10
6.7	Öðru vísi map fall	10

1 Viðfangaflutningar

Í forritunarmálum eru eru notuð fjögur til fimm afbrigði viðfangaflutninga (*parameter passing*), eftir því hvernig við teljum:

- Gildisviðföng (call by value).
- Tilvísunarviðföng (call by reference).
- Afritsviðföng (call by value-result).
- Nafnviðföng (call by name).
- Löt viðföng (call by need, lazy evaluation).

Í Pascal og C++ eru notuð gildisviðföng og tilvísunarviðföng. Í C, Java, Scheme og CAML eru aðeins gildisviðföng notuð. Í Ödu geta fyrstu þrjár aðferðirnar verið notaðar. Sum afbrigði FORTRAN nota bæði tilvísunarviðföng og afritsviðföng. Í Morpho eru gildisviðföng og einnig er hægt að líkja eftir tilvísunarviðföngum, nafnviðföngum og lötum viðföngum. Í Algol gamla voru gildisviðföng og nafnviðföng. Í Haskell eru löt viðföng. Einnig má halda því fram að λ -reikningur, sem fundinn var upp löngu áður en tölvur urðu til, noti nafnviðföng eða löt viðföng.

Ætlast er til að þið kunnið skil á viðfangaflutningum í þeim forritunarmálum sem notuð verða í námskeiðinu.

1.1 Gildisviðföng

Gildisviðfang (*call by value*) er gildað (*evaluated*) áður en kallað er á viðkomandi stef, gildið sem út kemur er sett á viðeigandi stað inn í nýju vakningarfærsluna (*activation record*) sem verður til við kallið.

Flest forritunarmál styðja gildisviðföng og við munum sjá þau í ýmsum forritunarmálum.

1.2 Tilvísunarviðföng

Tilvísunarviðfang (*call by reference*), t.d. `var` viðfang í Pascal eða viðfang með `&` í C++, verður að vera breyta eða ígildi breytu (t.d. stak í fylki). Það er ekki gildað áður en kallað er heldur er vistfang breytunnar sett á viðeigandi stað í nýju vakningarfærsluna. Þegar viðfangið er notað inni í stefinu sem kallað er á er gengið beint í viðkomandi minnissvæði, gegnum vistfangið sem sent var.

Við munum sjá tilvísunarviðföng í C++.

1.3 Afritsviðföng

Afritsviðfang (*call by value/result*, einnig kallað *copy-in/copy-out*) verður að vera breyta, eins og tilvísunarviðfang. Afritsviðfang er meðhöndlað eins og gildisviðfang, nema að þegar kalli lýkur er afritað til baka úr vakningarfærslunni aftur í breytuna.

Við munum ekki nota forritunarmál með afritsviðföngum.

1.4 Löt viðföng og nafnviðföng

Nafnviðföng virka þannig að þegar kallað er á fall eða stef er ekki reiknað úr nafnviðföngunum áður en byrjað er að reikna inni í fallinu eða stefinu sem kallað er á, heldur er reiknað úr hverju viðfangi í hvert skipti sem það er notað. Löt viðföng eru eins, nema hvað aðeins er reiknað einu sinni, í fyrsta skiptið sem viðfangið er notað. Ef nafnviðfang er breyta þá má nota það sem vinstri hlið í gildisveitingu. Hins vegar er ekkert vit í að nota latt viðfang sem vinstri hlið í gildisveitingu.

Við munum sjá nafnviðföng í Morpho og við munum sjá löt viðföng í Haskell og Morpho.

2 Dæmi

Í Morpho getum við skrifað eftirfarandi forritstexta, þar sem fallið `fg` notar gildisviðföng, fallið `ft` notar eftirlíkingu á tilvísunarviðföngum, fallið `fn` notar eftirlíkingu af nafnviðföngum og fallið `fl` notar eftirlíkingu af lötum viðföngum.

```
rec fun fg(x,y)
{
    x = x + 1;
    writeln("fg: "++x++ " "++y);
};
rec fun ft(&x,&y)
{
    x = x + 1;
    writeln("ft: "++x++ " "++y);
};
rec fun fn(@x,@y)
{
    x = x + 1;
    writeln("fn: "++x++ " "++y);
};
rec fun fl($x,$y)
{
    write("fl: ");
```

```

        writeln(x++ " " ++y);
    };
    rec fun id(n)
    {
        writeln("id");
        return n;
    };
    var a,b;
    b = 1;
    fg(b,b);
    writeln(b);
    b = 1;
    ft(&b,&b);
    writeln(b);
    b = 1;
    a = \ (1,2,3,4);
    fn(@b,@a[b]);
    writeln(b);
    val $d = $id(1);
    fl($d,$d);
    writeln(d);

```

Sé þessi forritstexti keyrður skrifast út eftirfarandi:

```

fg: 2 1
1
ft: 2 2
2
fn: 2 3
2
fl: id
1 1
1

```

Við sjáum hér dæmi um hvernig hægt er í Morpho að líkja eftir tilvísunarviðföngum, nafnviðföngum og lötum viðföngum. Öll viðföng í Morpho eru samt gildisviðföng og í öllum tilvikum hér er verið að senda gildi sem viðfang. Þetta er svipað og í C, sem aðeins hefur gildisviðföng, en sum þessara gildisviðfanga geta verið bendar, sem gerir okkur kleift að líkja eftir tilvísunarviðföngum í C.

Við munum kynnast Morpho betur síðar. Sækja má Morpho úr Canvas, en það er varla tímabært ennþá.

Í Scheme má einnig líkja eftir lötum viðföngum þegar við skilgreinum ný málfræðifyrirkærni í málinu, eins og við munum sjá, en annars notar Scheme alfarið gildisviðföng þegar um föll er að ræða.

3 Scheme

Við munum nú taka syrpu í að nota forritunarmálið Scheme vegna þess að það gefur okkur grundvöll til að tala um ýmis lykilatriði í merkingarfræði (*semantics*) forritunarmála almennt. Við munum annað slagið grípa til Scheme til að styrkja skilning okkar á ýmsu sem tengist merkingarfræði.

Ýmsar útfærslur af Scheme eru til, fyrir Windows, Linux og flest önnur stýrikerfi. Nefna má DrRacket (einnig kallað PLT Scheme og DrScheme) og MIT-Scheme, sem bæði eru til fyrir Windows, Linux og fleiri kerfi.

Auðvelt er að finna DrRacket á vefnum¹. DrRacket er meðal þægilegustu útgáfa af Scheme sem finna má, bæði í uppsetningu og notkun, þ.a. mælt er með henni. DrRacket er til á flest stýrikerfi. Þið getið einnig sótt MIT-Scheme af vefnum² og sett upp á eigin tölvum.

4 Forritspróun í Scheme

Ef við tökum MIT-Scheme sem dæmi (DrRacket má nota á svipaðan hátt, en einnig er auðvelt að nota DrRacket sem þróunarumhverfi (IDE)), þá getum við þróað forrit `fact.s` á eftirfarandi hátt:

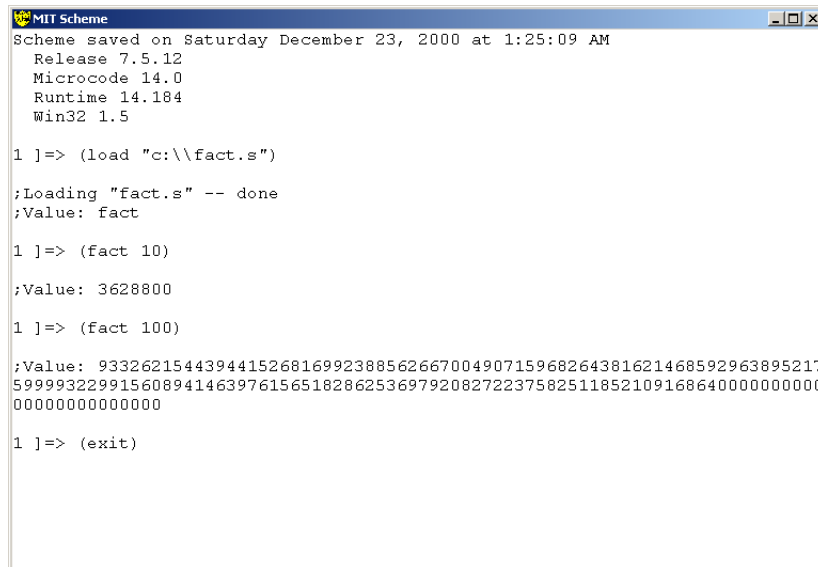
1. Ræsum Scheme úr valblaðsliðnum `Start → Programs → MIT Scheme → Scheme`.
2. Notum ritil til að skrifa eða breyta forritinu (skránni) `C:\Users\Nonni\TÖL304\fact.s`. (Væntanlega ekki Nonni, samt.)
3. Hlöðum forritinu í Scheme með skipuninni

```
(load "C:\\Users\\Nonni\\TÖL304\\fact.s")
```
4. Keyrum föll skilgreind í skránni, til prófunar. Ef villa finnst, förum við aftur í skref 2.
5. Þegar við erum orðin ánægð með innihald `C:\Users\Nonni\TÖL304\fact.s` hættum við í Scheme með því að nota skipunina (`exit`).

Mynd 1 sýnir dæmi um þetta.

¹<http://racket-lang.org/>

²<http://www.swiss.ai.mit.edu/projects/scheme/index.html>



```
MIT Scheme
Scheme saved on Saturday December 23, 2000 at 1:25:09 AM
Release 7.5.12
Microcode 14.0
Runtime 14.184
Win32 1.5

1 ]=> (load "c:\\fact.s")

;Loading "fact.s" -- done
;Value: fact

1 ]=> (fact 10)

;Value: 3628800

1 ]=> (fact 100)

;Value: 93326215443944152681699238856266700490715968264381621468592963895217
5999932299156089414639761565182862536979208272237582511852109168640000000000
0000000000000000

1 ]=> (exit)
```

Mynd 1: MIT Scheme í notkun

4.1 Ef þið notið DrRacket

Ef þið notið DrRacket, sem mælt er með, munið þá að stilla umhverfið á Scheme forritunarmálið með því að smella á Language→Choose Language og velja síðan **R5RS**.

5 Endurkvæmni og halaendurkvæmni

Í Scheme viljum við forrita án hliðarverkana, sem þýðir að við viljum ekki nota nein- ar gildisveitingar. Breytur fá því gildi þegar þær verða til og fá aldrei nýtt gildi. Þetta þýðir að forritunarstíll okkar breytist og við notum endurkvæmni (*recursion*) í stað lykkju. Endurkvæmt fall sem endar á að kalla á sjálft sig (eða jafnvel annað endurkvæmt fall) er kallað halaendurkvæmt (*tail recursive*). Scheme forritunarmálið er hannað þannig að halaendurkvæmni er sérstaklega hagstæð forritunaraðferð vegna þess að þegar Scheme fall endar á að kalla á annað fall (eða sjálft sig) er strax hætt í núverandi falli og næsta fall tekur við og skilar sínu gildi til þess sem kallaði á upp- haflega fallið. Þ.a. ef fall *f* kallar á fall *g* og fallið *g* endar á að kalla á fall *h*, þá mun vakning fallsins *g* gleymast um leið og kallað er á *h* og fallið *h* mun skila sínu gildi beint til fallsins *f* í stað þess að láta *h* skila til fallsins *g* sem síðan skili til *f*.

Mikilvægasta afleiðing af þessu er að djúp halaendurkvæmni étur ekki upp minni.

6 Nokkur Scheme föll

Í fyrirlestrum munum við ræða um fjölmörg Scheme föll. Hér eru nokkur á einfaldari nótunum.

Við munum gera mikla listavinnslu í Scheme og almennt í þessu námskeiði og við munum kappkosta að rökstyðja okkar forrit og föll. Á vefnum³ má finna Dafny útgáfur af ýmsum listavinnsluföllum sem vert er að kíkja á til að sjá annarsvegar mannlegan rökstuðning falla með athugasemdum og hins vegar rökstuðning sem Dafny kerfið samþykkir að sanni viðkomandi virkni. Ekki verður ætlast til þess að skrifaðar séu Dafny lausnir á prófi, en ætlast verður til að nemendur geti rökstutt á svipaðan hátt með athugasemdum, svipuðum og þeim sem sjá má á vefsíðunni og svipað eins og sjá má á þeim Scheme föllum sem sjá má hér að neðan.

Listavinnsla í Scheme byggist á föllunum `car`, `cdr` og `cons` sem uppfylla jöfnurnar

```
(car (cons x y)) == x
```

og

```
(cdr (cons x y)) == y
```

fyrir hvaða gildi `x` og `y` sem er. Einnig þurfum við sérstaka gildið `'()` sem stendur fyrir tóman lista. Þetta gildi er oftast kallað `null`, stundum `nil`, og er í eðli sínu svipað og `null` í Java. Í öðrum listavinnsluforritunarmálum eru samsvarandi föll, oftast með öðrum nöfnum.

Athugið að ef við höfum gildi

```
x=(cons x1 (cons x2 (cons ... (cons xN '()) ...)))
```

þá skrifum við það oftast sem

```
x=(x1 x2 ... xN)
```

sem er mun þægilegri ritháttur.

6.1 Lélegt *reverse*-fall

```
;; Notkun: (rev1 x)
;; Fyrir: x er listi (x1 ... xN)
;; Gildi: (xN ... x1)
(define (rev1 x)
  ;; Notkun: (append1 x y)
  ;; Fyrir: x er listi (x1 ... xN)
  ;; Gildi: (x1 ... xN y)
  (define (append1 x y)
    (if (null? x)
        (list y)
        (cons (car x) (append1 (cdr x) y)))))
```

³<https://rise4fun.com/Dafny/xR7n>

```

        (cons (car x) (append1 (cdr x) y))
    )
)
;; stofn fallins rev1:
(if (null? x)
    x
    (append1 (rev1 (cdr x)) (car x)))
)
)

```

6.2 Gott *reverse*-fall

Hér er halaendurkvæmni notuð til að ná fram lykkjuverkun og auka þannig hraðann.

```

;; Notkun: (rev2 x)
;; Fyrir: x er listi (x1 ... xN)
;; Gildi: (xN ... x1)
(define (rev2 x)
  ;; Notkun: (snuaskeyta x y)
  ;; Fyrir: x er listi (x1 ... xP),
  ;;        y er listi (y1 ... yQ)
  ;; Gildi: (xP ... x1 y1 ... yQ)
  (define (snuaskeyta x y)
    (if (null? x)
        y
        (snuaskeyta (cdr x) (cons (car x) y))))
  )
  (snuaskeyta x '())
)

```

6.3 Halaendurkvæmt Fibonacci-fall

Við reiknum með því að Fibonacci tölur F_0, F_1, \dots séu skilgreindar með

$$F_n = \begin{cases} 1 & \text{ef } n = 0 \text{ eða } n = 1 \\ F_{n-1} + F_{n-2} & \text{annars} \end{cases}$$

```

;; Notkun: (fibo n)
;; Fyrir: n er heiltala, >=0
;; Gildi: n-ta Fibonacci talan
(define (fibo n)
  ;; Notkun: (hjalp f1 f2 i)

```



```

;; Fyrir: 0 <= i <= n,
;;         f1 er i-ta Fibonacci talan,
;;         f2 er (i+1)-ta Fibonacci talan
;; Gildi: n-ta Fibonacci talan
(define (hjalp f1 f2 i)
  (if (= i n)
      f1
      (hjalp f2 (+ f1 f2) (+ i 1))
  )
)
;; stofn fallins fibo:
(hjalp 1 1 0)
)

```

6.4 „Venjulegt“ fall til að reikna n!

```

;; Notkun: (fact n)
;; Fyrir: n er heiltala, >=0
;; Gildi: n!
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))
)

```

6.5 Halaendurkvæmt fall til að reikna n!

```

;; Notkun: (fact n)
;; Fyrir: n er heiltala, >=0
;; Gildi: n!
(define (fact n)
  ;; Notkun: (hjalp n x)
  ;; Fyrir: n er heiltala, >=0, x er tala
  ;; Gildi: n!*x
  (define (hjalp n x)
    (if (= n 0)
        x
        (hjalp (- n 1) (* n x))
    )
  )
  (hjalp n 1)
)

```

6.6 Einfalt map fall

```
;; Notkun: (mymap f x)
;; Fyrir:  f er fall sem tekur eitt viðfang
;;         x er listi (x1 ... xN)
;; Gildi:  Listinn (y1 ... yN) þar sem
;;         yI er (f xI)
(define (mymap f x)
  (if (null? x)
      '()
      (cons (f (car x)) (mymap f (cdr x)))
  )
)
```

6.7 Öðru vísi map fall

Þetta map fall tekur fall sem viðfang og skilar falli.

```
;; Notkun: ((mymap2 f) x)
;; Fyrir:  f er fall sem tekur eitt viðfang,
;;         er er listi (x1 ... xN)
;; Gildi:  Listinn (y1 ... yN) þar sem
;;         yI er (f xI)
(define (mymap2 f)
  (lambda (x)
    (if (null? x)
        '()
        (cons (f (car x)) ((mymap2 f) (cdr x)))
    )
  )
)
```

Eða, jafngilt:

```
;; Notkun: ((mymap2 f) x)
;; Fyrir:  f er fall sem tekur eitt viðfang,
;;         er er listi (x1 ... xN)
;; Gildi:  Listinn (y1 ... yN) þar sem
;;         yI er (f xI)
(define (mymap2 f)
  (define (hjalp x)
    (if (null? x)
        '()
        (cons (f (car x)) (hjalp (cdr x)))
    )
  )
)
```

)
hja1p
)

Takið eftir því að innra fallið notar viðfangið f úr „efri földunarhæð“.