

# TÖL304G

## Forritunarmál

### Vikublað 1

Snorri Agnarsson

24. ágúst 2015

#### Efnisyfirlit

1	Inngangur	2
2	Hnit kennara	2
3	Lágmörk á verkefnaskilum	3
4	Miðmisserispróf	3
5	Dæmaskil og einkunnir	3
5.1	Snið lausna . . . . .	4
6	Piazza	5
7	Upptökur fyrirlestra	5
8	Helstu forritunarmál	5
9	Áhersluatriði	5
10	Bækur og annað lesefni	6
11	Áætlun	6
12	Efni vikunnar	7

13 Hvað er „mál“?	8
14 Samhengisfrjáls mál og BNF	8
15 Útleiðslur	8
16 Regluleg mál	9
17 Reglulegar segðir	10
18 Endanlegar stöðuvélar	10
19 Extended BNF	11
20 Dæmatímar og dæmahópar	12

## 1 Inngangur

Í þessu námskeiði verða kynntar grunnhugmyndir sem liggja að baki flestra forritunarmála. Tilgangurinn er meðal annars sá að nemendur geri sér grein fyrir mismun forritunarmála, notkunargildi hinna mismunandi mála og þeirra hugmynda, sem að baki liggja, og að nemendur verði dómbærir á kosti og galla hinna ýmsu forritunarmála.

Áhersluatriði eru betur skilgreind hér að neðan.

Bók til hliðsjónar er *Programming Languages* eftir Robert W. Sebesta.

Bókin er ætluð sem ítarefni en við munum ekki fylgja henni í neinum smáatriðum. Þeir sem hafa aðgang að öðrum bókum um forritunarmál geta örugglega notað þær í staðinn til hliðsjónar.

## 2 Hnit kennara

**Nafn:** Snorri Agnarsson

**Aðsetur:** Tæknigarður 231.

**Sími:** 861 3270

**Tölvupóstfang:** snorri@hi.is

### 3 Lágmark á verkefnaskilum

Samkvæmt reglum námsbrautarinnar á að skilgreina lágmark fyrir þau verkefni sem skilað hefur verið um miðbik misserisins. Þeir nemendur sem ekki ná því lágmarki á tilsettum tíma verða sjálfkrafa skráðir úr námskeiðinu og fá ekki að þreyta próf. Ef viðkomandi nemandi hefur gilda afsökun, svo sem veikindi, og getur sannfært kennarann um að hann eða hún hafi möguleika á að klára námskeiðið, þá má veita undanþágu frá þessari reglu.

Í þessu námskeiði þarf að skila að minnsta kosti átta verkefnum fyrir lok áttundu viku námskeiðsins. Í mörgum vikum, en ekki öllum, verður möguleiki á að skila tveimur verkefnum.

### 4 Miðmisserispróf

Um miðbik misserisins, sennilega í lok áttundu viku, verður miðmisserispróf sem gildir aðeins til hækkunar á lokaeinkun. Sé einkunnin úr miðmisserisprófinu hærri en lokaprófseinkunnin gildir miðmisseriseinkunnin 30% á móti lokaprófseinkunninni, sem gildir þá 70% af endanlegri prófseinkun.

### 5 Dæmaskil og einkunnir

Í námskeiðinu verða vikuleg einstaklingsverkefni og einnig verða oft hópverkefni til að vinna í dæmatíma. Það verða því vel yfir 13 lítil verkefni.

Öllum verkefnum, bæði hópverkefnum og einstaklingsverkefnum, skal skila fyrir miðnætti á fimmtudegi þeirrar viku sem verkefnin eru tengd við. Engin verkefni verða tengd fyrstu viku. Fyrstu verkefnaskilin verða tengd viku 2.

Engin takmörg eru sett á fjölda nemenda í hópi sem vinnur hópverkefni nema hvað nemandi má ekki vera í fleiri en einum skilahópi fyrir sama verkefni. Samræður milli hópa eru í fínu lagi.

Þar eð við munum nota Gradescope<sup>1</sup> kerfið til að fara yfir verkefni þá mun fjöldi stiga verða mismunandi fyrir hvert verkefni, en hins vegar verða allar verkefnaeinkunnir að lokum skalaðar á sama skala, þ.e. 0–10. Meðaltal 10 bestu einkunna verkefna mynda 30% af einkunn námskeiðsins. Öll verkefnaskil gilda jafnt, hvort sem þau eru hópverkefni eða einstaklingsverkefni.

Ekki er tekið á móti verkefnum eftir að skilafrestur er útrunninn.

Samvinna í vinnslu einstaklingsverkefna er alls ekki leyfð. Þó er í lagi að nemendur ræði saman sín á milli um dæmin og lausnir þeirra. En hver og einn nemandi skal skrifa (og prófa, ef hægt er) sína eigin lausn og skila henni. Afritun lausna eða hluta lausna er að sjálfsögðu forboðin með öllu.

---

<sup>1</sup><https://gradescope.com>

Verkefnum má skila á pappír í dæmahólf merkt Snorra Agnarssyni í anddyri VR-2 eða þeim má skila í tölvupósti til viðkomandi dæmakennara. Ef skilað er í tölvupósti er nauðsynlegt að „subject“ í skeytinu sé á sniði

TÖL304G, d1, einstaklingsverkefni 1: Jón Jónsson

eða

TÖL304G, d1, hópverkefni 1: Jón Jónsson

þar sem „d1“ er nafn dæmahópsins, „1“ er númer verkefnisins, og „Jón Jónsson“ er nafn nemandans sem er að skila.

Sé skilað í tölvupóst er algerlega nauðsynlegt að lausnin sé PDF skjal sem sett er sem viðhengi í tölvupóstinn.

Ef send eru skeyti þar sem „subject“ hefst ekki á „TÖL304G“ eða „Forritunarmál“ eða einhverju sem augljóstlega sýnir að skeytið fjallar um þetta námskeið er hætt við að skeytið verði meðhöndlað sem ruslpóstur og aldrei lesið.

## 5.1 Snið lausna

Vegna þess að við notum Gradescopeerfið til að fara yfir lausnir er nauðsynlegt að sérhver blaðsíða í lausninni innihaldi þær upplýsingar sem til þarf til að tengja lausnina við réttan nemanda.

Efst á sérhverri blaðsíðu skal skrifa nafn nemanda, háskólatölvupóstfang nemanda, nafn dæmahóps, nafn verkefnis, ásamt númeri viðkomandi blaðsíðu innan lausnar og heildarfjölda blaðsíðna í lausn. Til dæmis ef Jón Jónsson með háskólapóstfang jon123@hi.is skilar einstaklingsverkefni fyrir viku 2 á þremur blaðsíðum í dæmahópi 3 (annaðhvort á pappír eða í PDF skjali) þá skulu eftirfarandi línur vera efst á þessum þremur blaðsíðum, ein á hverri blaðsíðu:

Jón Jónsson, jon123@hi.is, einstaklingsverkefni 2, d3, 1/3

Jón Jónsson, jon123@hi.is, einstaklingsverkefni 2, d3, 2/3

Jón Jónsson, jon123@hi.is, einstaklingsverkefni 2, d3, 3/3

Sama gildir um hópverkefni, en þá skal einnig skrifa lista allra nemendanna í hópnum fremst á fyrstu blaðsíðu. Skrifa þarf bæði nafn og háskólatölvupóstfang hvers nemanda í hópnum því háskólatölvupóstfangið verður notað sem einkvæmur lykill í skráningu einkunna.

Sé skilað á pappír má nota báðar hliðar hvers blaðs ef það hentar, an athugið þá að að forhlið fyrsta blaðs verður síða 1, bakhliðin verður síða 2, o.s.frv. Pappírsverkefnið verða skönnuð og búnað til PDF skrár fyrir þau. Eftir á að koma í ljós hvernig gengur með Gradescope, en ef vel tekst til þá verður óþarfi að skila pappír til baka til nemenda því nemendur munu geta séð sín yfirlögn verkefni í vefkerfinu.

## 6 Piazza

Við munum nota spjallkerfið Piazza<sup>2</sup> til að skiptast á skoðunum um efni námskeiðsins. Allar almennar fyrirspurnir um námskeiðið eiga því heima þar. Ekki senda slíka fyrirspurnir í tölvupósti heldur beinið þeim til kennara, nemenda eða allra gegnum Piazza.

## 7 Upptökur fyrirlestra

Fyrirhugað er að taka fyrirlestra upp sem myndskleið sem verða aðgengileg gegnum Ugluna<sup>3</sup> í sérstökum flipa efst á síðu námskeiðsins.

## 8 Helstu forritunarmál

Forritunarmálin sem notuð verða eru trúlega Java, C++, C#, Scheme, Haskell, Morpho og CAML. Scheme er bálkmótað, einfalt en öflugt afbrigði af LISP. CAML er afbrigði af fallsforritunarmálinu ML. Haskell er hreint fallsforritunarmál, náskylt CAML, sem er í vaxandi notkun. Morpho er forritunarmál sem undirritaður hefur hannað og þróað með Hallgrími H. Gunnarssyni. Java þekkja flestir nú orðið. C# frá Microsoft er svipað og Java.

## 9 Áhersluatriði

Áhersla verður lögð á eftirfarandi atriði.

- Málfræði forritunarmála, BNF, EBNF og málrit. Einnig smávegis um regluleg mál, þ.e. reglulegar segðir og endanlegar stöðuvélar.
- Innviðir og hönnunargrundvöllur bálkmótaðra forritunarmála s.s. Ödu, Pascal, Morpho og Scheme.
- Viðfangaflutningar í forritunarmálum: Gildisviðföng, afritsviðföng, tilvísunarviðföng, nafnviðföng og löt viðföng.
- Grunnhugmyndir í fallforritun, löt og ströng gildun, hrein fallsforritun, kostir hennar og gallar.
- Einingaforritun í Morpho, Jövu og kannski C# og/eða C++. Sérstök áhersla verður lögð á fjölnota einingar (t.d. *template* í C++ og *generic* í Jövu).

---

<sup>2</sup>[www.piazza.com](http://www.piazza.com)

<sup>3</sup><http://ugla.hi.is>

- Skjölun forrita og sannprófun, einkum m.t.t. einingaforritunar og forritunar stórra kerfa.
- Listavinnsla í Scheme, Morpho, CAML og Haskell.
- Hlutbundin forritun í Java, C# og Scheme, áhersla er lögð á innviði hlutbundinnar forritunar, þ.e. hvernig boð vinna og samband þeirrar virkni við arfgengi.
- Fallsforritun í Scheme, CAML, Morpho og Haskell.
- Ruslasöfnunaraferðir, sem notaðar eru í málum s.s. Scheme, Morpho, Java, C#, CAML og Haskell.
- Kannski verður farið í samskeiða forritun í Morpho og Java.

Þau atriði, sem mest áhersla er lögð á að nemendur tileinki sér eru innviðir bálkmótaðra forritunarmála, einingaforritun, listavinnsla/fallsforritun (það tvennt hangir mjög saman), ruslasöfnun og hlutbundin forritun. Ekki er ætlast til að nemendur leggi á minnið smáatriði í málfræði forritunarmála. Áherslan er á merkingu forritunarmálanna og mismunandi eðli þeirra.

## 10 Bækur og annað lesefni

Bókin eftir Sebesta er til hliðsjónar í námskeiðinu og vísað verður í seinni vikublöðum á ýmsar hjálparskrár og vefsíður, til dæmis handbók fyrir forritunarmálið Morpho, auk rita um Scheme, Haskell, CAML og fleira.

Til dæmis er gagnlegt að líta á eftirfarandi vefsíður:

1. EBNF<sup>4</sup>
2. Veforðabók<sup>5</sup>

## 11 Áætlun

Líkleg efnisröð í námskeiðinu er eftirfarandi.

**Vika 1.** Málfræði forritunarmála, BNF, EBNF, málrit, endanlegar stöðuvélar, reglulegar segðir.

**Vika 2.** Viðfangaflutningur, forritunarmálið Scheme.

<sup>4</sup><http://www.cl.cam.ac.uk/mgk25/iso-ebnf.html>

<sup>5</sup><http://foldoc.org>

- Vika 3.** Scheme,  $\lambda$ -reikningur ( $\lambda$ -calculus), binding og sýnileiki nafna, báلكmótun (*block-structure*).
- Vika 4.** Vakningarfærslur (*activation records, stack frames*), lokanir (*closures*), framhöld (*continuations*).
- Vika 5.** Straumar í Scheme og einnig rökstudd forritun.
- Vika 6.** Forritunarmálið CAML (CAML Light eða Objective CAML).
- Vika 7.** Forritunarmálið Morpho, fjölnota einingar.
- Vika 8.** Meira Morpho, samskeiða forritun, hlutbundin forritun.
- Vika 9.** Miðannarpróf.
- Vika 10.** Lærum af miðannarprófinu, rifjum upp, fyllum í glufur.
- Vika 11.** Ruslasöfnunaraðferðir (*garbage collection methods*). Forritunarmálið Haskell.
- Vika 12.** Meira Haskell.
- Vika 13.** Fjölnota (*generic*) klasar í Jövu og C++.

## 12 Efni vikunnar

Í þessari og næstu viku förum við í mállýsingar forritunarmála og byrjum að kynna okkur innviði báلكbótaðra forritunarmála.

Í langflestum tilfellum er málfræði nútíma forritunarmála lýst með samhengisfrjáls-um mállýsingum, þ.e. BNF.

Oftast er málfræðinni lýst í tveimur skrefum, eitt skref lýsir frumeiningum málsins, þ.e. lykilorðum, sértáknum s.s. svigum, kommum, o.s.frv., og lesföstum s.s. heiltöluföstum og strengföstum. Í sama skrefi er oftast lýst því sem koma má milli frumeininga málsins, t.d. bilstafir og athugasemdir. Frumeiningum nútíma forritunarmála má langoftast lýsa sem reglulegum málum. Til dæmis er mál löglegra fleytitölufasta í C++ reglulegt mál.

Eftir að frumeiningum málsins hefur verið lýst er heildarmálfræði málsins lýst á BNF sniði. Þá er reiknað með því að frumeiningarnar, sem lýst var í fyrra skrefi séu tákn innan þess stafrófs, sem unnið er með.

## 13 Hvað er „mál“?

Mál er einfaldlega mengi strengja. Strengur er endanleg runa tákna úr einhverju mengi, sem við þá köllum táknróf (alphabet) málsins.

## 14 Samhengisfrjáls mál og BNF

BNF<sup>6</sup> er aðferð, svokallað meta-mál, til að skilgreina mál. Með BNF er í raun átt við það sama og átt er við þegar talað er um samhengisfrjálsar mállýsingar (context-free grammar).

BNF stendur nú fyrir Backus-Naur Form. BNF stóð einu sinni fyrir Backus Normal Form, því John Backus, sá sem fann upp FORTRAN forritunarmálið, átti mikinn þátt í þróun þess. En Peter Naur, sem ritstjóri skilgreiningarinnar á ALGOL-60 forritunarmálinu átti einnig mikinn þátt í að koma BNF í almenna notkun innan tölvunarfræðinnar, og þess vegna er BNF nú almennt látið standa fyrir Backus-Naur Form.

En BNF á sér einnig rætur í málvísindum, því málvísindamaðurinn Noam Chomsky skilgreindi samhengisfrjálsar mállýsingar áður en Backus or Naur skilgreindu BNF. Chomsky skilgreindi reyndar fleiri tegundir mállýsinga, m.a. fyrir regluleg mál, sem rætt er um hér að neðan.

Dæmi um BNF skilgreiningu:

```
<expr> ::= <num> | ( <expr> ) | <expr> <op> <expr>
<op> ::= + | - | * | /
<num> ::= <digit> | <digit> <num>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Þessi BNF skilgreining lýsir máli sem inniheldur strengi sem eru segðir („formúlur“) með heiltölugildum og venjulegum reikniaðgerðum.

## 15 Útleiðslur

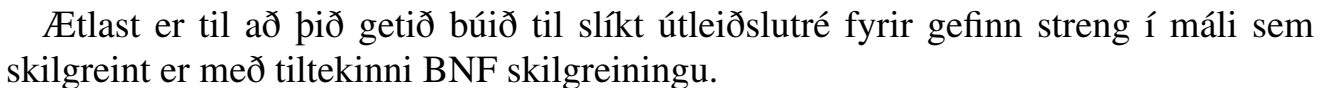
Þegar gefin er BNF skilgreining eins og að ofan má yfirleitt *leiða út* ótakmarkaðan fjölda strengja í málinu. Dæmi:

```
<expr>  =>
<expr> <op> <expr> =>
<expr> <op> <expr> <op> <expr>  =>
<num> <op> <expr> <op> <expr>  =>
<digit> <op> <expr> <op> <expr>  =>
1 <op> <expr> <op> <expr>  =>
```

<sup>6</sup><http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?Backus-Naur+Form>



Takið eftir að strenginn  $1 + 2 + 3$  má leiða út á fleiri en einn hátt. Mállýsing þessi er því *margræð*. Margræðni er óheppileg ef um forritunarmál er að ræða. Margræðnin sést vel ef við notum *útleiðslutré*:



Ef unnt er að lýsa tilteknu máli með BNF (þ.e. með samhengisfrjálsri mállýsingu) þá segjum við að *málið* sé samhengisfrjálst (ekki aðeins mállýsingin, sem augljóslega er samhengisfrjálst samkvæmt skilgreiningu).

Þegar málfræði forritunarmála er lýst er nú til dags næstum alltaf notuð einhver aðferð sem er jafngild BNF. Dæmi um slíkar aðferðir, aðrar en BNF sjálf, eru EBNF (Extended BNF) og málrít (syntax diagrams). Ætlast er til að þið getið lesið og skilið allar þessar þrjár aðferðir.

## 16 Regluleg mál

Frumeiningar forritunarmála, svo sem lykilorð, strengfastar, heiltölufastar, fleytitölufastar, o.s.frv., eru yfirleitt regluleg mál (regular language). Til dæmis er mál fleytitölufasta í Jövu reglulegt mál og mál strengfasta er annað reglulegt mál. Athugið að þetta eru að sjálfsögði tvö mismunandi regluleg mál, sem aftur eru notuð í skilgreiningu þriðja málsins (þ.e. Java).

Regluleg mál eru undirmengi samhengisfrjálsra mála, þ.e. öll regluleg mál eru samhengisfrjáls, en ekki öfugt. Regluleg mál eru einfaldari en önnur samhengisfrjáls mál, og til eru einfaldari aðferðir en BNF til að lýsa reglulegum málum. Ætlast er við að þið getið lesið og notað endanlegar stöðuvélar<sup>7</sup> (finite state automaton) og reglulegar segðir (regular expressions) til að lýsa einföldum reglulegum málum.

<sup>7</sup><http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=finite+state+automaton>

Regluleg mál eru mikið notuð í ýmsum tölvuverkefnum, til dæmis í skeljum og í leitarforritum svo sem grep.

## 17 Reglulegar segðir

Ein aðferð til að skilgreina regluleg mál er *reglulegar segðir*. Reglulegar segðir yfir stafróf  $\Sigma$  má skilgreinda á eftirfarandi hátt:

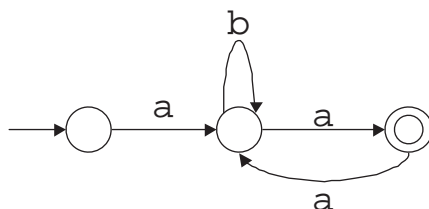
- $\epsilon$  er regluleg segð sem skilgreinir málið  $M(\epsilon) = \{\epsilon\}$
- Ef  $x \in \Sigma$  þá er  $x$  regluleg segð sem skilgreinir málið  $M(x) = \{x\}$ .
- Ef  $x$  og  $y$  eru reglulegar segðir þá er  $xy$  regluleg segð sem skilgreinir málið  $M(xy) = \{uv \mid u \in M(x) \wedge v \in M(y)\}$ .
- Ef  $x$  og  $y$  eru reglulegar segðir þá er  $x \mid y$  regluleg segð sem skilgreinir málið  $M(x \mid y) = \{w \mid w \in M(x) \vee w \in M(y)\}$ .
- Ef  $x$  er regluleg segð þá er  $x^?$  regluleg segð sem skilgreinir málið  $M(x^?) = M(x) \cup \{\epsilon\}$ .
- Ef  $x$  er regluleg segð þá er  $x^*$  regluleg segð sem skilgreinir málið  $M(x^*) = \bigcup_{n=0}^{\infty} M(x^n)$ .
- Ef  $x$  er regluleg segð þá er  $x^+$  regluleg segð sem skilgreinir málið  $M(x^+) = \bigcup_{n=1}^{\infty} M(x^n)$ .

Í ofangreindum skilgreiningum gerum við ráð fyrir að ef  $u$  og  $v$  eru strengir þá er strengurinn  $uv$  skilgreindur sem samskeyting  $u$  og  $v$ . Einnig gerum við ráð fyrir að veldishafning strengja sé skilgreind með eftirfarandi:

- $x^0 = \epsilon$
- $x^{n+1} = xx^n$  (skeytt er saman  $x$  og  $x^n$ ).

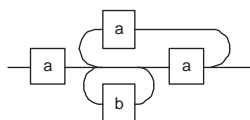
## 18 Endanlegar stöðuvélar

Eins og fram hefur komið eru endanlegar stöðuvélar enn ein aðferð til að skilgreina regluleg mál. Þessi mynd sýnir dæmi um endanlega stöðuvél sem ber kennsl á strengi yfir stafrófið  $\{a,b\}$ . Í málinu eru m.a. strengirnir  $aa$ ,  $aba$ ,  $abba$ , o.s.frv., en einnig má stinga  $aa$  einhvers staðar inn í löglegan streng og fá þannig út annan löglegan streng í málinu.



Strengi í málinu sem svona stöðuvél skilgreinir má fá með því að byrja í byrjunarstöðunni, sem er hringurinn sem örin lengst til vinstri bendir á, eða almennt sem sú ör bendir á sem ekki byrjar í einhverri stöðu, og fara síðan gegnum núll eða fleiri stikur og enda í lokastöðu. Lokastöður eru þær sem teiknaðar eru með tvöföldum hring. Ef síðan er skeytt saman þeim stöfum sem stikurnar eru merktar með þá fæst strengur í málinu. Aðrir strengir en þeir sem framleiða má með þessum hætti eru ekki í málinu.

Þessa endanlegu stöðuvél má einnig teikna sem málrít:



Takið eftir að í málrítnu eru kassar fyrir lokatáknin sem samsvara stikum í stöðuvélinni. Dæmi um strengi í þessu máli eru aa, aba, abba, aaaa, en ekki t.d. aaa eða aabaa.

Ætlast er til að nemendur geti skilið svona endanlegar stöðuvélar og málrít, og geti borið kennsl á hvort tilteknir strengir séu í málinu sem slíkar vélar og rit skilgreina. Fleiri dæmi um málrít má finna í handbókinni fyrir forritunarmálið Morpho, sem mun verða sett í Ugluna þegar þar að kemur.

## 19 Extended BNF

Extended BNF<sup>8</sup> eða EBNF er nú orðið algeng aðferð til að skilgreina málfræði forritunarmála. EBNF er svipað og BNF, en þar hefur verið bætt við hugmyndum úr reglulegum segðum. Ýmis afbrigði eru til af EBNF, eins og af BNF, en til er nú alþjóðlegur staðall<sup>9</sup> fyrir EBNF, sem líklegt er að flestir notendur EBNF fari nálægt að fara eftir. Sjá einnig grein eftir R.S. Scowen<sup>10</sup>.

Í EBNF eru lokatáknin merkt sérstaklega með því að setja gæsalappir utan um þau (andstætt venjunni í BNF, þar sem millitáknin eru merkt með <...>), en millitáknin eru venjuleg ómerkt orð eða jafnvel orðasambönd.

Í EBNF er síðan bætt við eftirfarandi málfyrirbærum í hægri hlutum reglna, sem eiga rætur að rekja til reglulegra segða:

<sup>8</sup>Sjá t.d. bls. 131-134 í Sebesta.

<sup>9</sup><http://www.cl.cam.ac.uk/mgk25/iso-ebnf.html>

<sup>10</sup><http://www.cl.cam.ac.uk/mgk25/iso-14977-paper.pdf>

- Notað má slaufusviga til að tákna endurtekningar:  $\{X\}$  er látið standa fyrir núll eða fleiri  $X$ .
- Notað má hornklofa til að tákna einstakan valkost:  $[X]$  er látið standa fyrir núll eða eitt  $X$ .
- Notað má sviga til að safna saman, og nota má  $|$  til að aðskilja valkosti: Til dæmis er  $(X | Y)Z$  jafngilt  $XZ | YZ$ .

Samkvæmt staðaltillögunni sem liggur fyrir er samskeyting í EBNF táknuð með aðgerðinni  $\rightarrow$ , (komma), sem er ólíkt BNF, þar sem samskeyting hefur ekkert aðferðartákn. Dæmi um einfalda EBNF mállýsingu er eftirfarandi:

```
expression =
    term, { op, term } ;
term =
    number | '(' , expression, ')' ;
op =
    '+' | '-' | '*' | '/' ;
number =
    digit { digit } ;
digit =
    '0' | '1' | '2' | '3' | '4' |
    '5' | '6' | '7' | '8' | '9' ;
```

Mállýsing þessi skilgreinir sama mál og BNF mállýsingin framar í þessu vikublaði. Einnig mætti lýsa sama máli með:

```
expression =
    ( number | '(' , expression, ')' ),
    { ( '+' | '-' | '*' | '/' ),
      ( number | '(' , expression, ')' ) } ;
number =
    ( '0' | '1' | '2' | '3' | '4' |
      '5' | '6' | '7' | '8' | '9' ),
    { '0' | '1' | '2' | '3' | '4' |
      '5' | '6' | '7' | '8' | '9' } ;
```

## 20 Dæmatímar og dæmahópar

Dæmatímar munu hefjast í viku 2.

Seinna verður send tilkynning um það hvaða dæmakennarar sjá um hvern dæmahóp. Skrifstofa deildarinnar mun væntanlega einnig leggja fram tillögu um hópaskiptingu.

Leyfilegt er að skipta um hóp, að því tilskildu að dæmakennarinn í hópnum sem þið viljið flytja til samþykki það. Dæmakennarar munu væntanlega samþykkja slíkar tilfærslur ef gild ástæða er til að flytja milli hópa og tilfærslurnar valda ekki óhóflegri stækkun einhvers hóps.

# TÖL304G

## Forritunarmál

### Vikublað 1

Snorri Agnarsson

30. ágúst 2015

## Efnisyfirlit

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

# 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  $\lambda$ -reikningur, sem fundinn var upp löngu áður en tölvur urðu til, noti nafnviðföng eða löt viðföng.

Ætlast er til að þið kunnið skil á viðfangaflutninum í þeim forritunarmálum sem notuð verða í námsskeiðinu.

## 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 síðari, en ég er ekki alveg tilbúinn til að eyða meiri tíma í það því fyrst þarf ég að uppfæra handbókina því ég gerði veigamiklar breytingar á málinu í sumar sem ekki eru enn allar komnar í handbókina. En sækja

má Morpho á vefnum<sup>1</sup>. Og hægt verður að sækja uppfærslur á sama stað þegar að því kemur að við viljum nota Morpho í verkefni.

Í Scheme má einnig líkja eftir lötum viðföngum þegar við skilgreinum ný málfræðifyrirkæri 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.

Þið getið sótt MIT-Scheme af vefnum<sup>2</sup> og sett upp á eigin tölvum. Einnig er auðvelt að finna DrRacket á vefnum<sup>3</sup>. 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.

### 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 rítill 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")
```

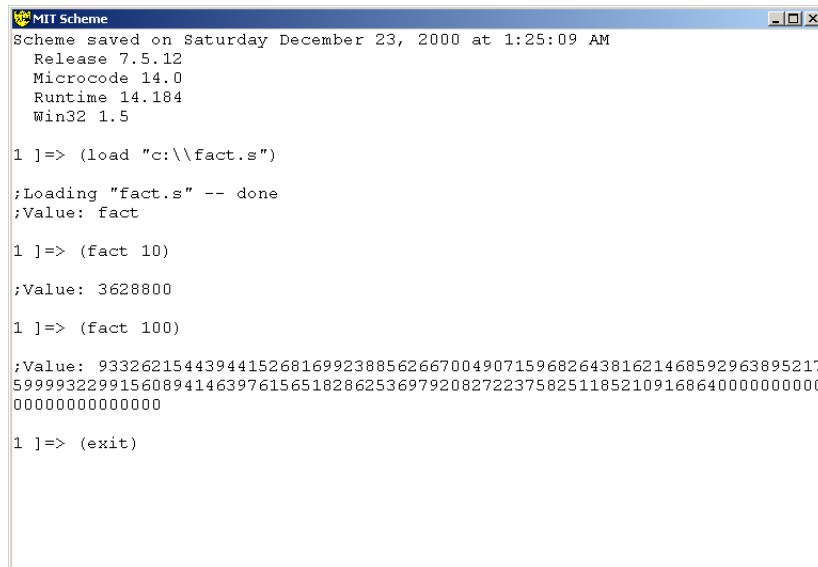
4. Keyrum föll skilgreind í skránni, til prófunar. Ef villa finnst, förum við aftur í skref 2.

---

<sup>1</sup><http://morpho.cs.hi.is>

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

<sup>3</sup><http://racket-lang.org/>



```
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

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.

## 4.1 Ef þið notið DrRacket

Ef þið notið DrRacket, 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.

### 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))))
  )
  )
  ;; stofn fallsins 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: (fibonacci n)
;; Fyrir: n er heiltala, >=0
;; Gildi: n-ta Fibonacci talan
(define (fibonacci 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 fallsins fibonacci:
(hjalp 1 1 0)
)
```

## 6.4 „Venjulegt“ fall til að reikna n!

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

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

# TÖL304G

## Forritunarmál

### Vikublað 3

Snorri Agnarsson

6. september 2015

## Efnisyfirlit

1	Efni vikunnar	1
2	Bindingar	2
3	Umdæmi	2
4	Lambda reikningur	2
4.1	Bindingar . . . . .	3
4.2	Innsetningar . . . . .	4
4.3	Reiknireglur . . . . .	4

## 1 Efni vikunnar

Í þessari viku höldum við áfram með Scheme og kíkjum einnig á bindingu og sýnileika nafna, sérstaklega í bálmótuðum forritunarmálum.

Nokkur lykilatriði í sambandi við bindingu og sýnileika eru:

- Skilgreiningar nafna.
- **Umdæmi** skilgreiningar (*scope*).
- **Földun** (*nesting*).
- **Földunarhæð** (*nesting level*).



- **Frjáls breyta** (*free variable*).
- **Bundin breyta** (*bound variable*).

## 2 Bindingar

Binding nafna í einhverju tilteknu máli er málefni, sem er mjög mikilvægt að þýðendur og notendur málsins séu sammála um.

Bindingar eru ekki aðeins mikilvægar í tölvufræðum, heldur jafnvel einnig í stærðfræðinni. Hver er t.d. merking formúlunnar  $\sum_{i=1}^{10} \sum_{i=1}^i i$ ? Getum við á skynsamlegan hátt sagt að þessi formúla hafi merkingu?

## 3 Umdæmi

Til þess að komast að niðurstöðu skilgreinum við **umdæmi** (*scope*) hverrar breytuskilgreiningar.

Í formúlu á sniðinu  $\sum_{i=X}^Y Z$ , þar sem  $i$  er breytunafn og  $X$ ,  $Y$  og  $Z$  eru formúlur, er breytan  $i$  skilgreind, og hefur væntanlega eitthvert vel skilgreint umdæmi, þ.e. það er væntanlega eitthvert vel skilgreint svæði innan formúlunnar þar sem  $i$  hefur þá merkingu, sem skilgreiningin gefur. Í þessu tilfelli er eðlilegt að skilgreina umdæmi þessarar breytu  $i$  sem undirformúluna  $Z$ .

Ef við vitum umdæmi tiltekinnar skilgreiningar eigum við að geta skipt um nafn á viðkomandi breytu án þess að merking formúlunnar breytist. Við megum gefa breytunni nýtt nafn ef það nafn kemur ekki fyrir annars staðar í formúlunni (reyndar ætti þessi regla að vera aðeins flóknari, en við komum að því síðar).

Með þessum bindingarreglum komumst við að því að formúlan að ofan er jafngild formúlunni  $\sum_{i=1}^{10} \sum_{j=1}^i j$ . Þessa niðurstöðu fáum við með því að skipta um nafn í undirformúlunni  $\sum_{i=1}^i i$  og fá jafngilda formúlu  $\sum_{j=1}^i j$ .

Takið eftir að innri summan í formúlunni  $\sum_{i=1}^{10} \sum_{i=1}^i i$  býr til **holu** í umdæmi ytri skilgreiningarinnar á  $i$ . Í hvert skipti sem breytunafn kemur fyrir í formúlu, hlýtur það að vísa til einnar og aðeins einnar skilgreiningar, og almenna reglan er sú að það sé sú skilgreining, sem er næst á undan í texta, eða næst fyrir utan í rúmi.

## 4 Lambda reikningur

Alonzo Church skilgreindi fyrir daga tölvunnar formúlur, sem kallast  $\lambda$ -formúlur (lambda formúlur), og reikninga með slíkar formúlur.

$\lambda$ -formúlur eru skilgreindar á eftirfarandi hátt:

- Ef  $x$  er breytunafn þá er  $x$  lögleg  $\lambda$ -formúla.

- Ef  $x$  er breytunafn og  $N$  er lögleg  $\lambda$ -formúla þá er  $\lambda x.N$  lögleg  $\lambda$ -formúla.
- Ef  $M$  og  $N$  eru löglegar  $\lambda$ -formúlur þá er  $MN$  lögleg  $\lambda$ -formúla.
- Ef  $M$  er lögleg  $\lambda$ -formúla þá er  $(M)$  lögleg  $\lambda$ -formúla.
- Engar aðrar formúlur eru löglegar  $\lambda$ -formúlur.

Við getum einnig lýst málinu á eftirfarandi hátt, þar sem við látum óskilgreint hvaða breytunöfn eru leyfð:

$$M \rightarrow x \mid (M) \mid MM \mid \lambda x.M$$

Mál þetta er margrætt, en við reiknum með því að leyst sé úr margræðninni með því að bæta svigum við eftir þörfum, þannig að ef  $M_1 M_2 M_3$  er  $\lambda$ -formúla sem samsett er úr minni  $\lambda$ -formúlum  $M_1$ ,  $M_2$  og  $M_3$  þá túlkum við hana sem jafngilda  $\lambda$ -formúlunni  $(M_1 M_2) M_3$ <sup>1</sup>. Hins vegar er  $\lambda x.M_1 M_2$  talin jafngild  $\lambda x.(M_1 M_2)$ .

## 4.1 Bindingar

Í  $\lambda$ -formúlum er skilgreind breytubinding, sem er svipuð þeirri bindingu, sem við þekkjum úr stærðfræðinni og forritunarmálum. Lykilhugtak þar er hvenær breytutilvísun er sögð vera *frjáls* í formúlu.

- Tilvísunin (*occurrence*) í breytuna  $x$  í  $\lambda$ -formúlunni  $x$  er frjáls.
- Ef  $N$  og  $M$  eru  $\lambda$ -formúlur þá eru allar tilvísanir í breytu  $x$  frjálsar í  $(NM)$ , sem eru frjálsar í  $N$  og  $M$ .
- Ef  $N$  er  $\lambda$ -formúla þá eru engar tilvísanir í breytuna  $x$  í  $\lambda$ -formúlunni  $\lambda x.N$  frjálsar, en aðrar tilvísanir, sem eru frjálsar í  $N$  eru frjálsar í  $\lambda x.N$ .

Breytutilvísun, sem ekki er frjáls, er sögð vera bundin. Breytan  $x$  er sögð vera bundin í undirformúlunni  $N$  í formúlunni  $\lambda x.N$ .

Einnig má skilgreina fallið *free*, sem tekur  $\lambda$ -formúlu sem viðfang og skilar mengi frjálsra breyta í formúlunni:

$$\begin{aligned} \text{free}(x) &= \{x\} \\ \text{free}(MN) &= \text{free}(M) \cup \text{free}(N) \\ \text{free}(\lambda x.M) &= \text{free}(M) - \{x\} \end{aligned}$$

<sup>1</sup>Mjög mikilvægt er að skilja þetta. Það er mikilvægur merkingarmunur á  $(M_1 M_2) M_3$  annars vegar og  $M_1 (M_2 M_3)$  hins vegar. Í fyrra tilfellinu er fallinu  $M_1$  beitt á viðfangið  $M_2$ , út úr því kemur fall sem er beitt á viðfangið  $M_3$ . Í seinna tilfellinu er fallinu  $M_2$  beitt á viðfangið  $M_3$ , út úr því kemur eitthvert gildi sem sent er sem viðfang í fallið  $M_1$ . Í Scheme væri þetta munurinn á segðunum  $((m1\ m2)\ m3)$  annars vegar og  $(m1\ (m2\ m3))$  hins vegar. Í Scheme, öfugt við  $\lambda$ -reiking, verðum við að setja svigana nákvæmlega svona.

## 4.2 Innsetningar

Í  $\lambda$ -reikningi eru skilgreindar **innsetningar** á formúlur, þar sem tilteknar frjálsar breytur fá „gildi“. Innsetningar má skrifa á sniðinu  $\{x_1 \rightarrow F_1, \dots, x_n \rightarrow F_n\}$ , og slíkri innsetningu má beita á  $\lambda$ -formúlu og fá út nýja  $\lambda$ -formúlu<sup>2</sup>. Formúlan  $\{x \rightarrow M\}N$ , þar sem  $M$  og  $N$  eru  $\lambda$ -formúlur, er ekki sjálf  $\lambda$ -formúla, en stendur fyrir þá  $\lambda$ -formúlu, sem út kemur þegar innsetningunni  $\{x \rightarrow M\}$  er beitt á  $N$ . Áhrif innsetninga með einni breytu eru skilgreind á eftirfarandi hátt:

- Ef  $x$  og  $y$  eru breytur,  $x \neq y$ , þá er  $\{x \rightarrow N\}y = y$ .
- $\{x \rightarrow N\}x = N$
- Ef  $L$ ,  $M$  og  $N$  eru  $\lambda$ -formúlur, þá er  $\{x \rightarrow L\}(MN) = (M'N')$ , þar sem  $M' = \{x \rightarrow L\}M$  og  $N' = \{x \rightarrow L\}N$ .
- Ef  $y$  er breyta,  $y$  er ekki  $x$ ,  $M$  og  $N$  er  $\lambda$ -formúla, þá er  $\{x \rightarrow N\}\lambda y.M = \lambda z'.M'$  þar sem  $z'$  er ný breyta, þ.e. ekki  $x$  og kemur ekki fyrir frjálts í  $N$  eða  $M$  og þar sem  $M' = \{x \rightarrow N\}\{y \rightarrow z'\}M$ .
- Ef  $M$  og  $N$  eru  $\lambda$ -formúlur, þá er  $\{x \rightarrow N\}\lambda x.M = \lambda x.M$ .

Það er að sjálfsögðu sterkt samband milli innsetninga og bindinga. Breytutilvísun er frjálts þá og því aðeins að innsetning hafi áhrif á hana.

## 4.3 Reiknireglur

Í  $\lambda$ -reikningi eru skilgreindar reiknireglur, sem lýsa því hvaða aðgerðir má gera á  $\lambda$ -formúlur án þess að breyta „gildi“ þeirra. Reglurnar eru eftirfarandi:

- ( $\beta$ -jafngildi) Ef  $N$  og  $M$  eru  $\lambda$ -formúlur þá má umskrifa  $(\lambda x.N)M$  sem  $\{x \rightarrow M\}N$ . Þessi regla samsvarar kalli á fall í forritun.
- Ef breyta  $y$  kemur ekki fyrir frjálts í  $N$  þá má umskrifa  $\lambda x.N$  sem  $\lambda y.N'$ , þar sem  $N' = \{x \rightarrow y\}N$ . Þessi regla samsvarar því þegar breytt er nafni á lepp í falli.

Takið eftir að í fyrri reglunni kemur ekki fram hvort búið er að „reikna út úr“ viðfanginu  $M$  áður en kallað er á fallið  $\lambda x.N$ . Það er reyndar svo í  $\lambda$ -reikningi að útkoman verður sú sama hvor leiðin sem farin er. Til dæmis getum við í  $\lambda$ -reikningi<sup>3</sup> skrifað bæði

$$(\lambda x.x^2)((\lambda y.(y+1))1) = (\lambda x.x^2)(1+1) = (\lambda x.x^2)2 = 2^2 = 4$$

<sup>2</sup>Takið eftir að oft er annar ritháttur, þ.e.  $\{N/x\}$  notaður fyrir innsetninguna  $\{x \rightarrow N\}$ , en hugmyndin er sú sama.

<sup>3</sup>Með örlitlum viðbótum við hreinan  $\lambda$ -reikning, til að leyfa aðeins flóknari formúlur, eins og einnig er gert í ýmsri umfjöllun um  $\lambda$ -reikning.

og

$$(\lambda x.x^2)((\lambda y.(y+1))1) = ((\lambda y.(y+1))1)^2 = (1+1)^2 = 2^2 = 4$$

Vegna þess að  $\lambda$ -reikningur leyfir ekki hliðarverkanir í föllunum er útkoman ávallt sú sama, hvor leiðin sem farin er (ef einhver endanleg útkoma fæst, sem er ekki alltaf).

# TÖL304G

## Forritunarmál

### Vikublað 4

Snorri Agnarsson

13. september 2015

## Efnisyfirlit

1	Vakningarfærslur	1
2	Lokanir	2
3	Framhöld	5
4	Straumar í Scheme	6

## 1 Vakningarfærslur

Vakningarfærslur (*activation records, stack frames*) í Scheme eru geymdar í kösinni (*heap*). Annars hefðum við ekki getað leyst öll verkefni á vikublaði 3. Við íhugum hvernig vakningarfærslur og lokanir (*closures*) eru meðhöndlaðar í Scheme og öðrum málum.

Vakningarfærsla er það minnissvæði sem einstök vakning af falli eða stafi notar meðan það keyrir. Öll forritunarmál sem hafa föll eða stef hafa vakningarfærslur á einn eða annan hátt. Í flestum tilfellum eru vakningarfærslur geymdar á hlaða (*stack*), en það er ekki algilt. Í öðrum tilfellum eru vakningarfærslur í kös (*heap*), t.d. í Scheme, Haskell, Morpho og ML, eða geymdar í sama minnissvæði og víðværar (*global*) breytur, t.d. í sumum afbrigðum af FORTRAN og COBOL. Þessi staðsetning vakningarfærslanna hefur afgerandi áhrif á notkunarmöguleika stefja og falla í viðkomandi forritunarmáli.

Í bálkmótuðum forritunarmálum (block-structured programming languages) innihalda vakningarfærslur (activation records) eftirfarandi upplýsingar:

- Viðföng (arguments).
- Staðværar breytur (local variables).
- Vendivistfang (return address).
- Stýrihlekk (dynamic link, control link).
- Tengihlekk (access link, static link).

Ef vakningarfærslur eru geymdar á hlaða, sem algengast er þegar þetta er ritað, bendir stýrihlekkurinn ávallt á næstu vakningarfærslu í hlaða, þ.e. vakningarfærslu þess stefs sem kallaði. Sama gildir í þeim fornu forritunarmálum sem einungis leyfðu mest eina samtímis vakningu á hverju falli.

Ef vakningarfærslur eru í kös má nota sömu aðferð með stýrihlekkinn, þ.e. láta hann ávallt benda á vakningarfærslu þess sem kallaði. En einnig kemur til greina að nota almennari aðferð sem gefur kost á almennri halaendurkvæmni, sem er sú aðferð að láta stýrihlekkinn benda á vakningarfærslu þess stefs sem á að fá niðurstöðuna úr núverandi kalli. Það stef þarf þá ekki endilega að vera stefið sem kallaði, þegar um halaendurkvæmni er að ræða. Vendivistfangið þarf þá að sjálfsögðu, til samræmis, að benda á viðeigandi stað í þulu (code) þess falls sem snúið er til baka í, þ.e. þess falls sem fá skal niðurstöðuna úr núverandi kalli.

Tengihlekkurinn bendir ávallt á viðeigandi vakningarfærslu þess stefs sem inniheldur viðkomandi stef, textalega séð. Viðeigandi vakningarfærsla er ávallt sú vakningarfærsla, sem inniheldur breytturnar í næstu földunarhæð, og er næstum alltaf nýjasta vakningarfærsla ytra stefisins. Undantekningar frá þeirri reglu geta orðið til við flókna notkun á *lokunum* (closures).

Vakningarfærslur í forritunarmálum, sem ekki eru bálkmótuð, s.s. C++, C# og Java, innihalda sömu upplýsingar og talið er upp að ofan, nema hvað tengihlekkur er ekki til staðar. Enda er tengihlekkur aðeins notaður til aðgangs að breytum í efri földunarhæðum, og er því merkingarlaus ef ekki er um bálkmótun að ræða.

## 2 Lokanir

Lokun (*closure*) er fyrirbæri, sem í bálkmótuðum málum er notað á sama hátt og fallsbendar eru notaðir í C og C++.

Í Standard Pascal eru lokanir til staðar, en ekki í Object Pascal (Delphi) eða Turbo Pascal. Dæmi um notkun og tilurð lokunar í Standard Pascal er eftirfarandi:

```

type func = function( x: Real ): Real;
function rot(f: func; a,b,eps: real): real;
begin
  if (b-a) < eps then
    rot := a
  else if f(a)*f((a+b)/2.0) <= 0 then
    rot := rot(f,a,(a+b)/2.0,eps)
  else
    rot := rot(f,(a+b)/2.0,b,eps)
end;

function h(a,b: real): real;
  var x,y,eps: real;
  function g(x: real): real;
  begin
    g:=a*x+b;
  end;
begin
  ... gefum x, y og eps viðeigandi gildi ...
  h:=rot(g,x,y,eps);
end;

```

Fallið sem er fyrsta viðfang í `rot` er lokun. Takið eftir að fallið `g` sem notað er sem viðfang í `rot` er faldað inn í fallið `h` og notar staðværar breytur í efri földunarhæð. Þær breytur (`a` og `b`) eru til staðar uns kallinu á `h` lýkur. Eftir að kallinu á `h` lýkur eru þær ekki lengur til.

Lokun inniheldur:

- Fallsbendi á vélarmálspulu viðkomandi falls.
- Aðgangshlekk, sem bendir á viðeigandi vakningarfærslu þess stefs, sem inniheldur viðkomandi fall.

Í eldri gerðum af bálkmótuðum forritunarmálum s.s. Standard Pascal er einungis hægt að senda lokanir niður sem viðfang í kall. Ekki er hægt að skila lokun sem niðurstöðu úr kalli eða vista lokun í breytu. Ástæða þessarar takmörkunar er sú að aðgangshlekkurinn í lokuninni inniheldur tilvísun á vakningarfærslu. Sú vakningarfærsla er áreiðanlega til staðar þegar lokunin verður til og allt þar til bálkur sá sem lokunin verður til í lýkur keyrslu, en eftir það er mögulegt að vakningarfærslunni sé eytt. Lokunin er aðeins í nothæfu ástandi ef aðgangshlekkurinn vísar á vakningarfærslu sem til er.

Í Standard Pascal er t.d. *ekki* löglegt að skrifa:

```

type adder = function( i: Integer ): Integer;
function newadder( k: Integer ): adder;
  function theadder( i: Integer ): Integer;
  begin
    theadder := k+i;
  end;
begin
  newadder := theadder; {þessi skipun gengur ekki}
end;

```

Nauðsynlegt er að nemendur skilji vel hvers vegna svona forrit eru ekki leyfð í Standard Pascal.

Í  $\lambda$ -reikningi er ekkert vandamál að skilgreina svona fall:

$$\text{newadder} = \lambda k.(\lambda i.i + k)$$

Í Object Pascal (Delphi) og gamla Turbo Pascal eru ekki lokanir. Í þeim forritunarmálum er ekki leyft að senda staðvær (*local*) föll sem viðföng, aðeins víðvær (*global*). Í Object Pascal má skrifa:

```

type func = function( x: Real ): Real;
function rot(f: func; a,b,eps: Real): Real;
begin
  if (b-a) < eps then
    rot := a
  else if f(a)*f((a+b)/2.0) <= 0 then
    rot := rot(f,a,(a+b)/2.0,eps)
  else
    rot := rot(f,(a+b)/2.0,b,eps)
end;

```

```

var globala, globalb: Real;

```

```

function g(x: Real): Real;
begin
  g:=globala*x+globalb;
end;

```

```

function h(a,b: real): real;
  var x,y,eps: real;
begin
  ... gefum x, y og eps viðeigandi gildi ...
  globala := a;

```



```

    globalb := b;
    h:=rot(g,x,y,eps);
end;

```

Þar má einnig skrifa:

```

type adder = function( i: Integer ): Integer;
var globalk: Integer;
function theadder( i: Integer ): Integer;
begin
    theadder := globalk+i;
end;
function newadder( k: Integer ): adder;
begin
    globalk := k;
    newadder := theadder; {þessi skipun gengur hér}
end;

```

Eins og sjá má leyfir Object Pascal að föll séu vistuð í breytum og að skilað sé föllum. En öll slík notkun á föllum takmarkast við víðvær föll. Slík fallsgildi eru *ekki* lokanir. Þar eð um víðvær föll er að ræða er engin þörf á aðgangshlekk.

Sum önnur bálkmótuð forritunarmál s.s. Scheme og ML hafa ekki þessa takmörkun á notkun lokana. Þau forritunarmál hafa ruslasöfnun (eins og Java, sem er ekki bálkmótað). Ruslasöfnun er nauðsynleg (en ekki nægjanleg) forsenda þess að unnt sé að nota lokanir á sveigjanlegan hátt. Og reyndar er það einnig nauðsynleg forsenda að vakningarfærslur taki þátt í ruslasöfnun. Í stað þess að vakningarfærslu sé skilað um leið og bálkur vakningarfærslunnar lýkur keyrslu lifir vakningarfærslan meðan til er einhver tilvísun á hana úr einhverjum lifandi lokunum.

### 3 Framhöld

Við höfum séð að lokun inniheldur tengihlekk og fallsbendi. Til er annað skylt fyrirbæri sem kallast framhald (*continuation*). Framhald inniheldur stýrihlekk og vendi-vistfang. Í Scheme má vinna með framhöld og í öðrum forritunarmálum má oft líta svo á að framhöld séu notuð í innviðum á útfærslum á afbrigðameðhöndlun, s.s. try-catch í Java og C++.

## 4 Straumar í Scheme

Á vefnum<sup>1</sup> má finna skjal um „óendanlega“ strauma í Scheme, ásamt Scheme forritstexta<sup>2</sup> fyrir föllin þar.

---

<sup>1</sup><http://www.hi.is/snorri/downloads/straumar.pdf>

<sup>2</sup><http://www.hi.is/snorri/downloads/straumar.s>

# TÖL304G

## Forritunarmál

### Vikublað 4

Snorri Agnarsson

21. september 2015

## Efnisyfirlit

### Efni vikunnar

Við höldum áfram með Scheme, lokanir, bálkmótun og strauma. Athugið að vakningarfærslur (*activation records*) eru lykillinn að skilningi á þessum fyrirbærum. Nauðsynlegt er að skilja hvernig vakningarfærslur eru notaðar, hvernig þær tengjast saman í keðjur gegnum stýrihlekki (*control link*, *dynamic link*) annars vegar, og gegnum tengihlekki (aðgangshlekki, *access link*, *static link*) hins vegar.

Einnig þarf að skilja hverjar afleiðingarnar eru af því að geyma vakningarfærslur á hlaða (*stack*), annars vegar, og í kös (*heap*), hins vegar. Ruslasöfnun minnis kemur einnig inn í dæmið, sem við munum sjá betur seinna.

### Rökstudd forritun

Finna má á vefnum<sup>1</sup> skjal um röksemdafærsluaðferðir í forritun. Þar eru meðal annars dæmi um notkun á ýmsum gerðum stöðulýsinga sem mikilvægar eru í rökstuddri forritun. Í þessu námskeiði verður gerð sú krafa að í verkefnum og á prófi séu úrlausnir skjalaðar með slíkum stöðulýsingum.

Krafan er sú að öll föll hafi lýsingu þar sem fram komi forskilyrði, eftirskilyrði og hvernig kalla skuli á fallið. Síðar, þegar við hefjumst handa við hlutbundna forritun

---

<sup>1</sup><http://www.hi.is/~snorri/downloads/rokjava.pdf>

og einingaforritun, þá verður krafist fastayrðingar gagna (data invariant) fyrir sérhvert nýtt gagnamót (data structure) sem skilgreint er.

# TÖL304G

## Forritunarmál

### Vikublað 6

Snorri Agnarsson

28. september 2015

## CAML Light og Objective CAML

Við munum kíkja á forritunarmálin CAML Light og Objective CAML, sem eru afbrigði af CAML, sem aftur er afbrigði af ML forritunarmálinu. Í framhaldinu munum við yfirleitt ekki gera greinarmun á CAML, CAML Light og Objective CAML.

Það sem einkennir ML og CAML er tögunin í þeim. Þau eru rammtöguð og nota sömu tögunaraðferð, sem er sérstök að því leyti að þýðandinn sér mikið til um að finna út úr því hvert tagið á að vera á hinum magvíslegustu gildum og segðum.

Til dæmis má nota eftirfarandi forritstexta í CAML til að skilgreina fall fyrir  $n!$ :

```
let rec fact n =  
  if n=0 then  
    1.0  
  else  
    (float_of_int n) *. (fact (n-1))  
;;
```

CAML þýðandinn mun lesa þessa skilgreiningu og draga þá ályktun að fallið `fact` sé af tagi `int -> float`. Þýðandinn sér að viðfangið `n` er af tagi `int` vegna þess að fallinu `float_of_int` er beitt á það, sem vitað er að tekur `int` sem viðfang (og skilar `float`), og útkoman úr `fact` er af tagi `float` vegna þess að lesfastinn `1.0` er af tagi `float` og fallið `*. skilar` ávallt `float`.

Ef við skrifum fallið svona

```
let rec fact n =  
  if n=0 then  
    1
```

```

else
    (float_of_int n) *. (fact (n-1))
;;

```

kvartar þýðandinn yfir því að fallið skili `int` (lesfastinn 1) á einum stað, en `float` á öðrum stað. Þýðandinn leyfir það ekki.

Öll föll í CAML taka *nákvæmlega eitt* viðfang af einhverju vel skilgreindu tagi og skila einu gildi af vel skilgreindu tagi. Þó er mikilvægt að hafa eitt lykilatriði í huga: Tagskilgreiningar í CAML (og ML) geta innihaldið *frjálsar tagbreytur*. Til dæmis er fallið `hd`, sem samsvarar `car` í LISP, af tagi `list 'a -> 'a`. Þetta þýðir að fallið tekur viðfang af tagi `list 'a`, þar sem `'a` stendur fyrir hvaða tag sem er, og skilar þá gildi af tagi `'a`.

Vegna tögunarinnar er í CAML gerður greinarmunur á pörum og listum, sem ekki er gert í Morpho og LISP (þ.m.t. Scheme), ef viðkomandi gildi er ekki tómur listinn. Ef til dæmis `'a` og `'b` eru tvö tög þá er `'a * 'b` tag þar sem fyrra stakið er af tagi `'a` og seinna af tagi `'b`. Aftur á móti er `list 'a` tag lista gilda af tagi `'a`. Slíkur listi má vera tómur, en það getur þar ekki verið.

CAML má finna á vefsíðum frönsku rannsóknarstofnunarinnar INRIA<sup>1</sup>. Þar má fá CAML í ýmsum útgáfum og útfærslum, bæði CAML Light og Objective CAML, sem nú heitir OCAML, fyrir ýmis stýrikerfi. Mælt er með CAML Light fyrir þetta námskeið, en ef menn ætla að nota CAML fyrir bitastæð verkefni er næstum örugglega betra að nota OCAML. Þeir sem áhuga hafa mega vel nota OCAML í námskeiðinu.

## Hvar finnum við CAML Light?

CAML kerfið fyrir ýmis stýrikerfi má sækja af vefnum<sup>2</sup>.

## Nokkur CAML Light föll

Athugið að flest þessi föll eru reyndar einnig innbyggð í CAML Light.

### Haus lista

```

(*
** Notkun: hd x
** Fyrir: x er listi, ekki tómur
** Gildi: Hausinn á x, þ.e. fremsta gildið
*)
let hd x =

```

<sup>1</sup><http://caml.inria.fr/download.en.html>

<sup>2</sup><http://caml.inria.fr/caml-light/index.en.html>

```

match x with
  [] ->
    raise
      (Invalid_argument
        "attempt to take head of empty list"
      )
  |
    a::b ->
      a
;;
(* hd : 'a list -> 'a = <fun> *)

```

## Hali lista

```

(*
** Notkun: tl x
** Fyrir: x er listi, ekki tómur
** Gildi: Halinn á x
*)
let tl x =
  match x with
    [] ->
      raise
        (Invalid_argument
          "attempt to take tail of empty list"
        )
    |
      a::b ->
        b
;;
(* tl : 'a list -> 'a list = <fun> *)

```

## Innsetning frá vinstri

```

(*
Notkun: it_list f u x
Fyrir: f er tvíundaraðgerð,  $f: A \rightarrow B \rightarrow A$ ,
       u er gildi af tagi A,
        $x=[x_1; \dots; x_N]$  er listi gilda af
       tagi B.
Gildi:  $u+x_1+x_2+\dots+x_N$ , reiknað frá vinstri
       til hægri, þar sem  $a+b = (f\ a\ b)$ .
*)

```

```

let rec it_list f u x =
  if x=[] then
    u
  else
    it_list f (f u (hd x)) (tl x)
;;

(*
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
*)

```

## Innsetning frá hægri

```

(*
Notkun: list_it f x u
Fyrir: f er tvíundaraðgerð, f: A->B->B,
       u er gildi af tagi B,
       x=[x1;...;xN] er listi gilda af
       tagi A.
Gildi: x1+x2+...+xN+u, reiknað frá hægri
       til vinstri, þar sem a+b = (f a b).
*)
let rec list_it f x u =
  if x=[] then
    u
  else
    f (hd x) (list_it f (tl x) u)
;;

(*
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
*)

```

## Beiting falls á lista

```

(*
** Notkun: map f x
** Fyrir: x er 'a list = [x1;x2;...;xN], f er 'a->'b
** Eftir: y er listinn [f x1;f x2;...;f xN]
*)
let rec map f x =
  if x = [] then
    []

```



```

    else
      (f (hd x)) :: (map f (tl x))
;;
(* map : ('a -> 'b) -> 'a list -> 'b list = <fun> *)

```

## Viðsnúningur lista

```

let reverse x =
  let rec revapp x y =
    if x = [] then
      y
    else
      revapp (tl x) ((hd x) :: y)
  in
    revapp x []
;;
(* reverse : 'a list -> 'a list = <fun> *)

```

## Samskeyting lista

```

let rec append x y =
  if x=[] then
    y
  else
    (hd x) :: (append (tl x) y)
;;
(* append : 'a list -> 'a list -> 'a list = <fun> *)

```

## Y fallið

```

(*
** Notkun: y f
** Fyrir: f er fall sem tekur lélegt fall af
**        tagi 'a -> 'b sem viðfang og skilar
**        betra falli
** Gildi: Besta fall 'a -> 'b sem unnt er að
**        fá með því að endurbæta gagnslaust
**        fall með f
*)
let rec y f x =
  (f (y f)) x
;;
(* y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)

```

## Dæmi um notkun Y

Eftirfarandi segð reiknar  $10!$  með því að nota endurbætingarfall.

```
let bf f n =  
  if n = 0 then  
    1.0  
  else  
    (float_of_int n) *. f(n-1)  
in  
  (Y bf) 10  
;;  
(* - : float = 3628800 *)
```

# TÖL304G

## Forritunarmál

### Vikublað 7

Snorri Agnarsson

4. október 2015

## Efnisyfirlit

<b>1 Morpho</b>	<b>1</b>
1.1 Morpho keyrsluumhverfið . . . . .	2
<b>2 Notkun Morpho</b>	<b>5</b>

## 1 Morpho

Við byrjum nú að leggja stund á Morpho. Aðalatriðin í notkun Morpho eru listavinnsla og einingaforritun. Listavinnslu höfum við kynnst áður í Scheme og CAML. Listavinnslan í Morpho er eins, en þó er sá munur að í Morpho er venjan frekar sú að nota hliðarverkanir þegar henta þykir. Til dæmis notum við lykkjur óspart, og snúningur lista í Morpho gæti verið forritaður á eftirfarandi hátt:

```
1 z=x; y=[];
2 ;;; x=z=[x1,...,xN], y=[]
3 while( z!=[] )
4 {
5     ;;; Fastayrðing: z=[xI,...,xN], y=[xI-1,...,x1]
6     y = head(z) : y;
7     z = tail(z);
8 };
9 ;;; y=[xN,...,x1]
```

Þetta má forrita í einingu á eftirfarandi hátt:

```

1 "reversion.mmod" =
2 {{
3   ;;; Notkun: z = reverse(x)
4   ;;; Fyrir: x er listi [x1,...,xN]
5   ;;; Eftir: z er nýr listi [Xn,...,x1]
6 reverse =
7   fun(x)
8   {
9     var z=x, y=[];
10    while( z!=[] )
11    {
12      ;;; z=[xI,...,xN]
13      ;;; y=[xI-1,...,x1]
14      y = head(z) : y;
15      z = tail(z);
16    };
17    return y;
18  };
19 }};

```

Einnig má forrita þetta sem staðvært fall:

```

1   ;;; Notkun: z = reverse(x)
2   ;;; Fyrir: x er listi [x1,...,xN]
3   ;;; Eftir: z er nýr listi [Xn,...,x1]
4   rec fun reverse(x)
5   {
6     var z=x, y=[];
7     while( z!=[] )
8     {
9       ;;; z=[xI,...,xN]
10      ;;; y=[xI-1,...,x1]
11      y = head(z) : y;
12      z = tail(z);
13    };
14    return y;
15  };

```

Morphopýðandann og meðfylgjandi skrár má finna í ZIP skrá á vefnum<sup>1</sup>. Afpakk-ið þessari skrá í möppu á ykkar tölvu. Einu skrárnar þar sem eru bráðnauðsynlegar eru `morpho.jar`, sem er bæði pýðandi og keyrsluumhverfi fyrir Morpho, og `Morpho.pdf`, sem er handbók fyrir Morpho. Morpho er frjáls til afnota fyrir alla.

## 1.1 Morpho keyrsluumhverfið

Morpho er báلكmótað forritunarmál með lokunum og samhliða vinnslu. Þetta flækir dálítið málið þegar kemur að hönnun keyrsluumhverfisins. Mynd 1 sýnir almennt

<sup>1</sup><http://morpho.cs.hi.is/morpho/dist/morphodist.zip>

ástand í keyrslu Morpho sýndarvélarinnar.

Við ræðum um hana í fyrirlestri til að styrkja skilninginn á því hvernig innviðir forritunarmála virka.

Í keyrslu Morpho sýndarvélarinnar eru nokkur gisti (*register*) í gangi.

**code.** Fylki af Morpho vélarmálsskipunum sem verið er að framkvæma.

**pc.** Vísir inn í **code** sem bendir á þá vélarmálsskipun sem verið er að framkvæma. Samanlagt skilgreina **code** og **pc** staðsetningu í vélarmálspulu sem líta má að sé sambærilegt við vendivistfang eða fallsbendi í forritunarmálum sem hafa einfaldara keyrsluumhverfi.

**stack.** Keðja af hlekkjum sem hver og einn inniheldur eina breytu. Þessi gildahlaði er umhverfið (*environment*) sem verið er að keyra í. Hér eru bæði staðværar breytur og breytur í efri földunarhæðum.

**ret.** Eðlilegt framhald úr núverandi falli (sjá neðar).

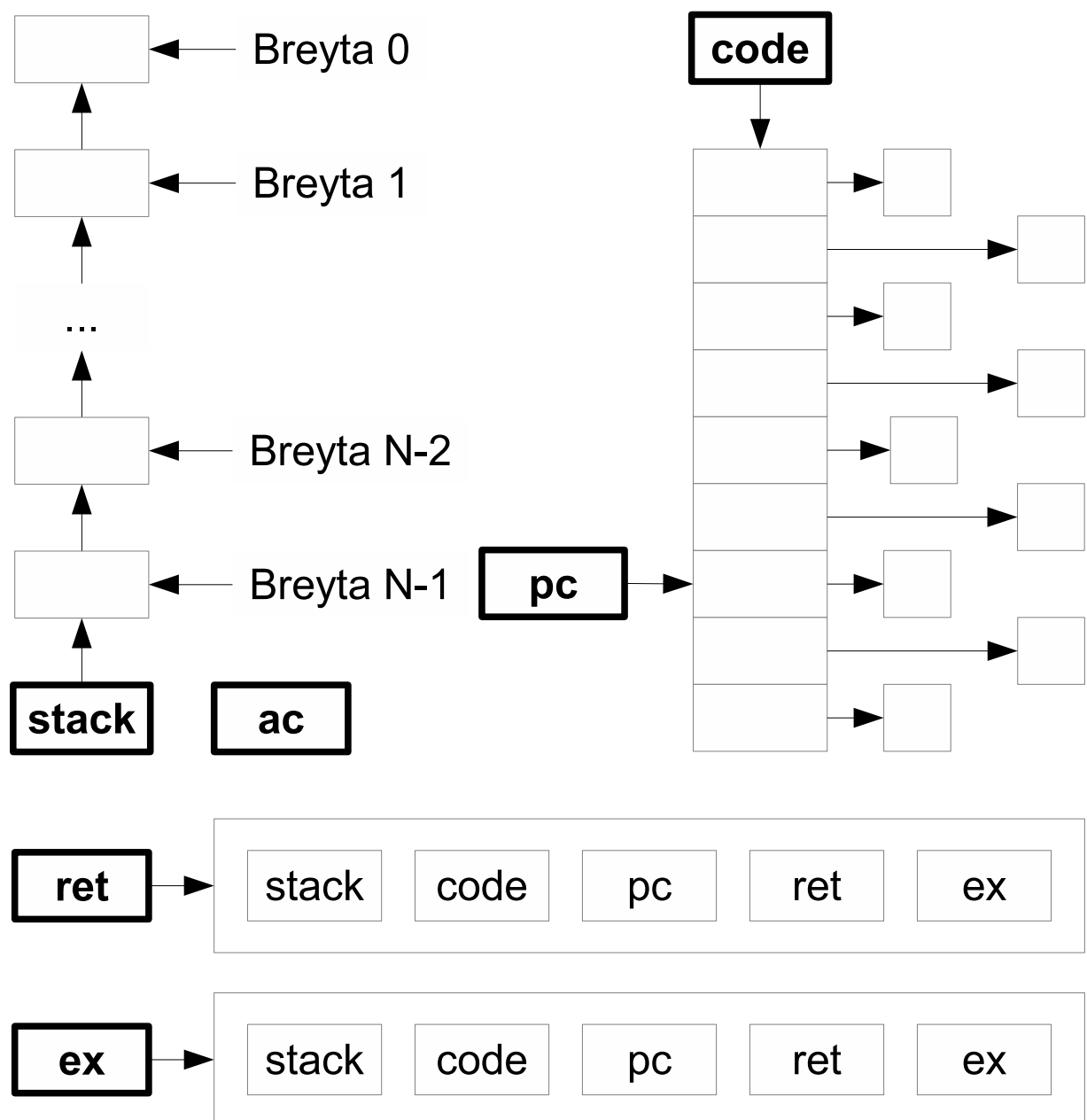
**ex.** Afbrigðilegt framhald úr núverandi falli (sjá neðar).

**ac.** Gisti sem fær nýtt gildi í hvert sinn sem Morpho segð skilar gildi (*accumulator*). Þegar fall skilar gildi er gildið sett í **ac** áður en snúið er til baka úr fallinu.

Eitt lykilatriði í Morpho, og einnig í Scheme og mörgum svipuðum málum, er að þegar við köllum á fall þá sendum við fallinu bæði viðföng (á hlaðanum) og **framhald** (*continuation* á ensku). Framhaldið er í grundvallaratriðum bendir á vélarmálspulu ásamt bendi á vakningarfærslu. En sama gildir um lokanir og hver er þá munurinn? Almennt er framhald gildi sem er dálítið keimlíkt og lokun en í stað þess að innihalda bendi á vakningarfærslu sem verður í næstu földunarhæð þegar lokunin er nytjuð (kallað er á lokunina), þá inniheldur framhald bendi á vakningarfærslu sem verður núverandi vakningarfærsla þegar framhaldið er nytjað (snúið er til baka úr núverandi falli). Halaendurkvæmni er þá útfærð með því að halaendurkvæmt kall fær sama framhald og sá sem kallar, í stað þess að búið sé til nýtt framhald til að snúa til baka til þess sem kallar. Lykilatriði er að framhaldið inniheldur stýrihlekk í stað þess að innihalda aðgangshlekk eins og lokun gerir.

Reyndar er ein afleiðing af þessari hönnun að aðgangshlekkir eru ekki lengur bendar á vakningarfærslur heldur eru bendar á hlekki í gildahlaðanum. Þetta veldur minnis-sparnaði þegar breytur þurfa að lifa það af að fallið deyr sem bjó þær til, en þið þurfið ekki að muna það smáatriði og megið reikna með því að aðgangshlekkir séu bendar á vakningarfærslur. Í þessari hönnun getum við litið svo á að heil vakningarfærsla sé samanlagt innihaldið í gistunum **stack**, **ret** og **ex**.

Á hverju andartaki í keyrslu Morpho eru tvö framhöld tiltæk. Annað framhaldið er í gistinu **ret** og er eðlilegt framhald sem nytjað er þegar núverandi fall skilar gildi. Takið eftir að framhaldið inniheldur allt sem til þarf til að stilla sýndarvélina í skilgreint



Mynd 1: Staða í keyrslu Morpho

ástand nema gildið í gistingu **ac**. En þegar snúið er til baka úr kalli þarf fallið einmitt að tilgreina gildið sem fer í **ac** gistið.

Hitt framhaldið er í gistingu **ex**, sem nytjað er þegar afbrigði gerist í keyrslu (þ.e. *exception*). Þá er gildi afbrigðisins (oft Java Exception eða Error gildi) sett í **ac** gistið og hægt að að grípa það í `catch` hluta af `try-catch` segð.

## 2 Notkun Morpho

Keyra má Morpho í samtalsham (*interactive mode*) með skipuninni

---

```
morpho
```

---

eða (jafngilt)

---

```
java -jar morpho.jar
```

---

Til að þýða Morphoforrit úr textaskrá `x.morpho` má nota eftirfarandi skipun (í Windows):

---

```
morpho -c x.morpho
```

---

eða (í Unix):

---

```
./morpho -c x.morpho
```

---

eða (bæði í Windows og Unix):

---

```
java -jar morpho.jar -c x.morpho
```

---

Nánari upplýsingar er að finna í handbókinni fyrir Morpho.

# TÖL304G

## Forritunarmál

### Vikublað 8

Snorri Agnarsson

11. október 2015

## Efnisyfirlit

1	Miðannarpróf	1
2	Upplýsingahuld og hönnunarskjöl	3

## 1 Miðannarpróf

Fimmtudaginn 22. október verður miðannarpróf. Einkunnin í miðannarprófinu gildir til hækkunar á prófseinkun í námskeiðinu, ef einkunnin er hærri en einkunnin á loka-prófi. Í því tilfelli gildir miðannarprófseinkunnin 30% á móti lokaprófseinkuninni.

Efnið fyrir miðannarprófið er allt sem rætt hefur verið um fram að prófinu, en meðal mikilvægustu efnisatriða eru eftirfarandi:

- Mállýsingar
  - BNF, EBNF og málrít
  - Regluleg mál, reglulegar segðir, endanlegar stöðuvélar, löggengar og brigðgengar.
- Viðfangaf lutningar
  - Gildisviðföng (call by value)
  - Tilvísunarviðföng (call by reference)
  - Afritsviðföng (call by value-result)



- Nafnviðföng (call by name)
- Löt viðföng (lazy evaluation)
- Bálmótun og vakningarfærslur (*block structure, activation records*)
  - Innihald vakningarfærslsna (*activation record*) með og án bálmótunar
    - \* Viðföng
    - \* Staðværar breytur
    - \* Aðgangshlekkur (aðeins í bálmótuðum)
    - \* Vendivistfang
    - \* Stýrihlekkur

Viðföng, staðværar breytur og aðgangshlekkur mynda **umhverfi** (*environment*), og vendivistfang ásamt stýrihlekk mynda **framhald** (*continuation*).

- Aðgangshlekkir (tengihlekkir) (*access link, static link*)
- Stýrihlekkir (*control link, dynamic link*)
- Lokanir (*closure*)
- Skilgreiningar nafna
- Umdæmi skilgreiningar (*scope*)
- Földun (*nesting*)
- Földunarhæð (*nesting level*)
- Frjáls breyta (*free variable*)
- Bundin breyta (*bound variable*)
- Fallsforritun (*function programming*)
  - Forritun „án hliðarverkana“ í Scheme og CAML
  - Endurkvæmni og halaendurkvæmni
  - Æðra stigs föll: Föll sem skila föllum
  - Listavinnsla (*list processing*)
- Einingaforritun og rökstudd forritun
  - Upplýsingahuld
  - Stöðulýsingar, þar með talið sérstaklega fastayrðingar gagna
  - Notkunarlýsingar: „Notkun:“, „Fyrir:“ og „Eftir:“
  - Samband hönnunar, smíðar, notkunarlýsinga og fastayrðingar gagna

- Tögun án breytuyfirlýsinga (CAML)
  - Ætlast er til að þið getið greint tag einfaldra falla í CAML.
- Scheme
  - Báلكmótað fallsforritunarmál með keyrslutögun
  - Afbrigði af LISP
- CAML
  - Báلكmótað fallsforritunarmál með rammtögun
  - Tagað án breytuyfirlýsinga
- Morpho
  - Báلكmótað einingaforritunarmál með listavinnslu

Prófið verður án hjálpargagna og athugið að líta skal á öll heimaverkefni sem efni til prófs.

## 2 Upplýsingahuld og hönnunarskjöl

D.L. Parnas setti fyrir löngu síðan fram þá meginreglu í hugbúnaðarsmíð að sérhverja veigamikla hönnunarákvörðun skyldi fela í einingu. Til dæmis, ef nota skal forgangsbiðraðir í kerfinu skal fela í einingu hvernig forgangsbiðraðirnar eru útfærðar. Tilgangurinn með þessu er að sjálfsögðu sá að unnt verði að breyta ákvörðuninni seinna án þess að það kosti stóran uppskurð á kerfinu.

Í forritun í stórum stíl (*programming in the large*) er skynsamlegt að líta á einingu frá þremur sjónarhornum, frá sjónarhorni *notenda*, *smíða* og *hönnuðar*.

Eðlilegt er að gera ráð fyrir að fleiri en einn aðili sé notandi sömu einingar, að einingin sé notuð í fleiri en einu kerfi.

Einnig er eðlilegt að gera ráð fyrir að fleiri en einn aðili sé smíður sams konar einingar, að eining sé smíðuð oftari en einu sinni án þess að munur sé á notkun einingarinnar.

Hins vegar er eðlilegt að gera ráð fyrir að aðeins einn aðili sé hönnuður einingar. Hönnun einingar felst þá í því að skrifa lýsingu einingarinnar, sem innihaldi allar þær sameiginlegu upplýsingar fyrir notendur og smíði sem nauðsynlegar eru til að nota eða smíða eininguna, en ekki meiri upplýsingar.

Hönnunarskjal skal uppfylla eftirfarandi skilyrði:

- Gefa skal notanda einingar nægilega miklar upplýsingar um smíð einingarinnar til að hann geti notað hana, *en ekki meiri upplýsingar*.
- Gefa skal smíð einingar nægilega miklar upplýsingar um notkun einingarinnar til að hann geti smíðað hana, *en ekki meiri upplýsingar*.

# TÖL304G

## Forritunarmál

### Vikublað 8

Snorri Agnarsson

19. október 2015

## Efnisyfirlit

<b>1</b>	<b>Miðannarpróf</b>	<b>1</b>
<b>2</b>	<b>Gamlir fyrirlestra í Uglu</b>	<b>2</b>
<b>3</b>	<b>Verkefni</b>	<b>2</b>
<b>4</b>	<b>Hlutbundin forritun í Morpho</b>	<b>2</b>
4.1	Einfaldur hlutbundinn hlaði . . . . .	2
4.2	Hlaði með víxlunarboði . . . . .	4
4.3	Hlaðaeining með innfluttum arfi . . . . .	5
4.4	Samband boða og aðferða . . . . .	7
4.5	Hlaði með uppnefndum boðnöfnum . . . . .	8

## 1 Miðannarpróf

Miðannarprófið verður fimmtudaginn 22. október í fyrirlestratímanum.

Prófið verður án hjálpargagna og athugið að líta skal á öll heimaverkefni sem efni til prófs. Prófið verður eilítið öðru vísi en próf fyrri ára því ekki verður neitt val milli spurninga.

## 2 Gamlir fyrirlestra í Uglu

Þar eð fyrirlesturinn mánudaginn 19. október fellur næstum örugglega niður verður sett upptaka af fyrirlestri á svipuðum nótum frá því í fyrra í Ugluna. Nemendur ættu að taka sér tíma til að horfa á hann.

## 3 Verkefni

Verkefnablað vikunnar verður á venjulegum stað í Uglunni.

## 4 Hlutbundin forritun í Morpho

### 4.1 Einfaldur hlutbundinn hlaði

Dæmi um klasaskilgreiningu í Morpho er eftirfarandi eining

```
1  ;;; Hönnun
2  ;;;
3  ;;; Útflutt
4  ;;;
5  ;;; Notkun: s = stack();
6  ;;; Fyrir: Ekkert.
7  ;;; Eftir: s er nýr tómur hlaði með pláss
8  ;;;        fyrir ótakmarkaðan fjölda gilda
9  ;;;        meðan minnisrými tölvunnar leyfir.
10 ;;;
11 ;;; Innflutt
12 ;;;
13 ;;; Notkun: s.push(x);
14 ;;; Fyrir: s er hlaði.
15 ;;; Eftir: Búið er að setja x ofan á x.
16 ;;;
17 ;;; Notkun: x = s.pop();
18 ;;; Fyrir: s er hlaði, ekki tómur.
19 ;;; Eftir: x er gildið sem var efst á s,
20 ;;;        það hefur verið fjarlæggt af s.
21 ;;;
22 ;;; Notkun: b = s.isEmpty();
23 ;;; Fyrir: s er hlaði.
24 ;;; Eftir: b er true ef s er tómur, annars
25 ;;;        false.
```

```

26
27 "stack.mmod" =
28 {{
29 stack =
30   obj()
31   {
32     var list;
33
34     ;;; Fastayrðing gagna: Hlaði sem inniheldur gildi
35     ;;; x1,...,xN, frá toppi til botns er táknaður
36     ;;; með list = [x1,...,xN].
37
38     msg push(x)
39     {
40       list = x:list;
41     };
42
43     msg pop()
44     {
45       val res = head(list);
46       list = tail(list);
47       res;
48     };
49
50     msg isEmpty()
51     {
52       list == [];
53     };
54   };
55 }}
56 ;

```

Þessi forritstexti varpar ljósi á nokkur grundvallaratriði:

- Einingin sem skilgreind er hér hefur eitt útflutt stef, `stack`. Það stef er smiður (*constructor*) fyrir hluti sem eru hlaðar, þ.e. þeir hlutir bregðast rétt við boðunum `push`, `pop` og `isEmpty`.
- Takið eftir að breytan `list` er tilviksbreyta.
- Takið eftir að í aðferðum boðanna er bæði unnt að nota tilviksbreytuna og viðföngin sem send eru í boðin.
- Vert er að taka fram að tilviksbreytu er aðeins unnt að nota í aðferðum sem

fylgja smíð þess klasa sem inniheldur tilviksbreytuna. Ekki er hægt að nota tilviksbreytuna utan frá né í hlutum sem erfa frá viðkomandi hlut.

## 4.2 Hlaði með víxlunarboði

Síðan getum við haldið áfram og búið til annan klasa sem erfir frá þessum:

```
1  ;;; Hönnun
2  ;;;
3  ;;;   Útflutt
4  ;;;
5  ;;;   Notkun: h := stack()
6  ;;;   Eftir: h er nýr tómur aukinn hlaði
7  ;;;
8  ;;;   Innflutt
9  ;;;
10 ;;;   Notkun: h.swap()
11 ;;;   Fyrir: h er aukinn hlaði með a.m.k. tvö gildi
12 ;;;   Eftir: búið er að víxla efstu tveimur gildunum
13 ;;;           á h
14 ;;;
15 ;;;   Einnig eru boðin push, pop og isEmpty
16 ;;;   innflutt, og hafa sömu lýsingar og fyrir
17 ;;;   stack.mmod.
18
19 "stack2.mmod" =
20 {{
21 stack =
22   obj() super(stack())
23   {
24     msg swap()
25     {
26       var x,y;
27       x = this.pop();
28       y = this.pop();
29       this.push(x);
30       this.push(y);
31     };
32   };
33 }}
34 *
35 "stack.mmod"
```

Í þessu dæmi kemur fram að lykilorðið `this` stendur fyrir hlut þann sem viðkomandi aðferð er verið að framkvæma í. Lykilorð `this` má aðeins nota inni í aðferð fyrir boð. Þá er lykilorðið `super`, þegar það kemur fyrir í haus hlutskilgreiningar, notað til að tilgreina klasa sem erft er frá. Stefkallið á eftir lykilorðinu (í þessu dæmi kallið `stack()`) verður að skila hlut.

### 4.3 Hlaðaeining með innfluttum arfi

Einnig er hægt að búa til almennari fjölnota klasa sem erfir frá innfluttum klasa:

```

1  ;;; Hönnun
2  ;;;   Útflutt
3  ;;;
4  ;;;   Notkun: s = stack()
5  ;;;   Eftir: s er nýr tómur aukinn erfður hlaði,
6  ;;;   þar sem erfði hlaðinn kemur úr
7  ;;;   innflutta stefinu stack, sem lýst
8  ;;;   er að neðan, og aukningin felst
9  ;;;   í því að s hefur aðferðir fyrir
10 ;;;   boðin height og maxHeight, sem lýst
11 ;;;   er að neðan
12 ;;;
13 ;;;   Innflutt
14 ;;;
15 ;;;   Notkun: s = stack()
16 ;;;   Eftir: s er erfður hlaði. Erfði hlaðinn má
17 ;;;   ekki hafa önnur boð en push og pop
18 ;;;   sem breyta fjölda gilda á hlaðanum
19 ;;;
20 ;;;   Notkun: n = s.height()
21 ;;;   Fyrir: s er aukinn erfður hlaði
22 ;;;   Eftir: n er fjöldi gilda á s
23 ;;;
24 ;;;   Notkun: n = s.maxHeight()
25 ;;;   Fyrir: s er aukinn erfður hlaði
26 ;;;   Eftir: n er mesti fjöldi gilda sem verið hafa
27 ;;;   samtímis á s
28 ;;;
29 ;;;   Auk þess eru boðin push og pop innflutt og
30 ;;;   hafa sömu lýsingu og í erfða hlaðanum.
31
```

```

32 "stackstat.mmod" =
33 {{
34 stack =
35   obj() super(stack())
36   {
37     var count=0, max=0;
38     ;;; Fastayrðing gagna:
39     ;;;   count er fjöldi gilda á hlaðanum,
40     ;;;   max er hámarksfjöldi gilda sem verið hafa
41     ;;;   samtímis á hlaðanum.
42
43     msg height()
44     {
45       count;
46     };
47     msg maxHeight()
48     {
49       max;
50     };
51     msg push(x)
52     {
53       count = inc(count);
54       count > max && (max = count);
55       super.push(x);
56     };
57     msg pop()
58     {
59       count = dec(count);
60       super.pop();
61     };
62   };
63 }}
64 ;

```

Hér sést að lykilorðið `super` er einnig notað inni í aðferðum til að tiltaka að kalla skuli á erfða aðferð fyrir tiltekið boð. Takið eftir að það er merkingarmunur á segðunum `this.pop()` og `super.pop()`.

Fyrri segðin kallar á „venjulegu“ `pop` aðferðina fyrir hlutinn, þ.e. þá aðferð sem er fremst í arfgengiskeðjunni. Seinni segðin kallar á þá `pop` aðferð sem er fyrir aftan núverandi aðferð í arfgengiskeðjunni. Þetta er alvanalegt í hlutbundinni forritun, svipað eins og í C++, Java og flestöllum öðrum hlutbundnum forritunarmálum.

Vert er að benda sérstaklega á að eininguna `"stackstat.mmod"` má nota með



hvaða hlaðaklasa sem er, sem hagar sér eins og klasarnir í "stack.mmod" og "stack2.mmod". Til dæmis má skrifa:

```
"stack3.mmod" = "stackstat.mmod" * "stack.mmod" ;  
og  
"stack4.mmod" = "stackstat.mmod" * "stack2.mmod" ;
```

## 4.4 Samband boða og aðferða

Í öllum hlutbundnum forritunarmálum fylgir hverjum hlut vörpun sem varpar boði í aðferð. Oftast er þetta gert þannig að hverjum klasa fylgir slík vörpun, og oft er vörpunin útfærð sem einfalt fylki þar sem vísirinn er heiltala sem stendur fyrir boðið og gildið er fallsbendir eða lokun sem stendur fyrir aðferðina. Þetta fylki er kallað á ensku *Virtual Method Table*, skammstafað VMT. Við getum kallað þetta fylki *aðferðatöflu* á íslensku.

Í Morpho er aðferðatafla fyrir hvern klasa sem búinn er til með hlutstefi, þ.e. stafi á sniðinu

```
f =  
  obj (...) ...  
  {  
    ...  
  }
```

Þegar boð er sent til hlutar er leitað að boðinu í aðferðatöflunni og ef aðferð finnst fyrir boðið er hún framkvæmd. Ef ekki þá er athugað hvort hluturinn erfir frá einhverjum öðrum hlut. Ef svo er þá er leitað að aðferðinni í erfða hlutnum, og svo koll af kolli. Ef engin aðferð finnst fyrir boðið þá er það villa. Slík villa getur að sjálfsögðu ekki gerst í rammtöguðum forritunarmálum, en í Morpho, Fjölni, SmallTalk og öðrum keyrslutöguðum hlutbundnum forritunarmálum getur það gerst.

Í töguðum hlutbundnum forritunarmálum svo sem Java, C++ og Object Pascal er, eins og rætt hefur verið, yfirleitt farið aðeins öðruvísi að. Þá fylgir hverjum klasa aðeins ein aðferðatafla, jafnvel þótt um arfgengi sé að ræða. Sú aðferðatafla varpar boði beint í aðferð, hvort sem aðferðin er útfærð í viðkomandi klasa eða í yfirklasa. Þetta er ekki hægt í Morpho því yfirklassi er ekki þekktur á þýðingartíma, og er reyndar ekki þekktur fyrr en hluturinn verður til.

Í öllum tilfellum er boð því tilvísun á aðferð, þegar gefinn er hlutur eða klasi. Boð er ekki aðferð því mismunandi hlutir hafa mismunandi aðferðir fyrir sama boð. Það er einmitt stóri kosturinn við hlutbundna forritun að sama boð getur haft mismunandi aðferðir í mismunandi hlutum.

## 4.5 Hlaði með uppnefndum boðnöfnum

Ef nú svo vill til að nauðsynlegt reynist að nota hlaða þar sem nöfn stefja og boða eru á íslensku þá má nota eftirfarandi:

```
1  ;;; Hönnun
2  ;;;   Útflutt
3  ;;;
4  ;;;   Notkun: h = hlaði();
5  ;;;   Eftir: h er nýr tómur hlaði
6  ;;;
7  ;;;   Innflutt
8  ;;;   Boðin setja , sækja og erTómur eru innflutt og
9  ;;;   hafa sömu virkni og samsvarandi boð push , pop
10 ;;;   og isEmpty í einingunni "stack.mmod".
11
12 "hlaði.mmod" =
13 {{
14 hlaði = fun stack();
15 }}
16 *
17 "stack.mmod"
18 *
19 {{
20 push = msg setja(x);
21 pop = msg sækja();
22 isEmpty = msg erTómur();
23 }}
24 ;
25
26 ;;; Prófunarforrit fyrir hlaða
27 "hladaþrof.mexe" = aðal in
28 {{
29 aðal =
30     fun()
31     {
32         var x = [1,2,3,4];
33         var h = hlaði();
34         while( x )
35         {
36             h.setja(head(x));
37             x = tail(x);
38         };
```

```
39         while ( !h.erTómur() )
40         {
41             writeln(h.sækja());
42         };
43     };
44 }}
45 *
46 "hladi.mmod"
47 *
48 BASIS
49 ;
```

# TÖL304G

## Forritunarmál

### Vikublað 10

Snorri Agnarsson

26. október 2015

## Efnisyfirlit

<b>1 Efni vikunnar</b>	<b>1</b>
<b>2 Haskell</b>	<b>2</b>
<b>3 Náms efni í Haskell</b>	<b>2</b>
3.1 Vefgæði . . . . .	2
3.2 Löt gildun . . . . .	2
3.3 Listaritháttur . . . . .	3
<b>4 Haskell dæmi</b>	<b>5</b>
4.1 Prímtölur . . . . .	5
4.2 Pýþagórasarprenndir . . . . .	6

## 1 Efni vikunnar

Við byrjum á að ræða *ruslasöfnun*, eftir að við förum yfir prófið.

Í framhaldi af því byrjum við síðan að ræða um forritunarmálið Haskell.

Í Uglunni má finna bækling um ruslasöfnun sem þið skuluð lesa.

## 2 Haskell

Forritunarmálið Haskell er tagað fallsforritunarmál með latrí gildun. Haskell hefur heimasíðu á vefnum<sup>1</sup> og sækja má Haskell þýðendur þaðan. Við munum nota Glasgow Haskell<sup>2</sup>.

## 3 Námsefni í Haskell

Í Haskell munum við sjá ýmislegt nýtt:

- Löt gildun (lazy evaluation, call by need).
- Nýr listaritháttur (list comprehension), sem minnir á hefðbundinn rithátt fyrir mengi í stærðfræðinni.
- Ýmsir mismunandi valkostir til að skilgreina jafngild föll (t.d. pattern matching).
- Ný aðferð til að forrita með hliðarverkunum án þess að glata aðalkostum fallsforritunar (*einstæður*, monads).

Af þessum atriðum eru það lata gildunin og listarithátturinn sem við munum leggja aðaláherslu á.

Auk þess hefur Haskell tögun sem er mjög lík töguninni í CAML. Í Haskell eru reyndar nýjir möguleikar í töguninni sem CAML býður ekki upp á, en ekki verður lögð áhersla á þá þætti.

### 3.1 Vefgæði

Matthías Páll Gissurarson benti mér einu sinni á góða Haskell bók á vefnum, Learn You a Haskell for Great Good!<sup>3</sup>.

Einnig má finna vefsíðuna Try Haskell<sup>4</sup> þar sem prófa má Haskell án þess að setja neitt upp á tölvunni þinni.

### 3.2 Löt gildun

Í Haskell er það ófrávíkjanleg regla að viðföng falla eru ekki gilduð (evaluated) fyrr en nauðsyn krefur. Sama gildir um listaskilgreiningar, sem hefur þær afleiðingar að listar í Haskell eru keimlíkir straumum í Scheme.

Eftirfarandi Haskell skilgreiningar nýta sér lötu gildunina í Haskell:

---

<sup>1</sup><http://www.haskell.org>

<sup>2</sup><http://www.haskell.org/platform/>

<sup>3</sup><http://learnyouahaskell.com>

<sup>4</sup><http://tryhaskell.org>

```

1  and :: [Bool] -> Bool
2  and [] = True
3  and (True : xs) = and xs
4  and (False : xs) = False
5
6  or :: [Bool] -> Bool
7  or [] = False
8  or (True : xs) = True
9  or (False : xs) = or xs
10
11 fib1 :: [Integer]
12 fib1 =
13     [1,1] ++ map (\(a,b)->a+b) (zip fib1 (tail fib1))
14
15 fib2 :: [Integer]
16 fib2 = [1,1] ++ zipWith (+) fib2 (tail fib2)

```

Lesið ykkur til um föllin map, zip og zipWith sem hér eru notuð.

Lata gildunin í Haskell veldur stundum vandræðum. Íhugið til dæmis þetta fall:

```

1  sum f n =
2      hjaalp 0 0
3      where
4          hjaalp s i =
5              if i>n then
6                  s
7              else
8                  hjaalp (s+f(i+1)) (i+1)

```

Þegar við köllum á þetta fall með Haskell segðinni

```
1  sum (\i->i^2) 10
```

þá reiknum við út  $\sum_{i=1}^{10} i^2$ .

En það gerist ekki með því að fyrst sé reiknuð talan 0, síðan talan  $0 + 1^2$ , síðan talan  $0 + 1^2 + 2^2$ , o.s.frv.

Það sem gerist er að fyrst er smíðuð *segðin* 0, síðan *segðin*  $0 + 1^2$ , síðan *segðin*  $0 + 1^2 + 2^2$ , o.s.frv. Að lokum höfum við segðina  $0 + 1^2 + 2^2 + \dots + 10^2$ , sem verður þá loksins gilduð til að skila gildi Haskell segðarinnar **sum** (\i->i^2) 10.

### 3.3 Listaritháttur

Haskell listarithátturinn hefur (auk venjulegs ritháttar, svipað og í CAML) annars vegar einfaldar runur á fernu sniði:

- [i ..]
- [i .. j]
- [i,j ..]

- `[i,j .. k]`

og hins vegar flóknari og öflugri rithátt sem býður upp á skilgreiningar svo sem þessar:

```
1 fib3 :: [Integer]
2 fib3 = [1,1] ++ [a+b | (a,b) <- zip fib3 (tail fib3)]
```

Þessi listaritháttur í Haskell er kallaður *list comprehension* og er hannaður til að vera svipaður í útliti og merkingu eins og mengjarithátturinn sem við eigum að venjast.

Við erum vön að sjá mengjaskilgreiningar svo sem  $\{x^2 | x \in \{1..5\}\}$ . Í Haskell má á svipaðan hátt skilgreina *listann*

```
1 [ x^2 | x <- [1..5] ]
```

Þetta er lögleg Haskell segð sem skilar `[1,4,9,16,25]`.

Athugið þó að listasegð er e.t.v. ekki reiknanleg. Til dæmis er gagnslaust að skilgreina eftirfarandi lista, þótt löglegur sé:

```
1 [(x,y,z) | x <- [1..], y <- [1..], z <- [1..], x^2+y^2==z^2]
```

En skrifa má aftur á móti:

```
1 [(x,y,z) | x <- [2..],
2           y <- [1..(x-1)],
3           let z=floor $ sqrt $ fromIntegral (x^2+y^2),
4           x^2+y^2==z^2,
5           (gcd (gcd x y) z)==1
6 ]
```

Íhugið þessa segð vandlega. Hér kemur fyrir allt það sem leyft er í þessari gerð lista-skilgreininga í Haskell:

- Framleiðendur (*generators*) svo sem

`x <- [2..]`.

- Skilgreiningar svo sem

`let z=floor $ sqrt $ fromIntegral (x^2+y^2)`

- Síur (*filters*) svo sem

`x^2+y^2==z^2`.

Á Haskell síðunni<sup>5</sup> má finna formlega skilgreiningu á listarithættinum.

Listarithátturinn bætir engu við það sem forrita má í Haskell án hans. Með hjálp fallanna `concatMap`, `map` og `filter` má gera allt það sem gera má með listarithættinum. En það er þá oftast flóknara og krefst stundum auk þess einfaldra hjálparfalla.

Til dæmis:

<sup>5</sup><http://haskell.org/onlinereport/exps.html>

```

1  [x^2|x<-[1..10]]
2  =
3  map (\x->x^2) [1..10]
4  =
5  concatMap (\x->[x^2]) [1..10]

og:

1  [(x,y)|x<-[1,2,3],y<-['a','b','c']]
2  =
3  concatMap (\x->[(x,y)|y<-['a','b','c']]) [1,2,3]
4  =
5  concatMap (\x->(concatMap (\y->[(x,y)])
6                        ['a','b','c']
7                        ))
8                        )
9                        [1,2,3]

```

## 4 Haskell dæmi

Kíkið á eftirfarandi Haskell dæmi.

### 4.1 Prímtölur

Listi (straumur) allra prímtalna.

```

1  — Höfundur: Snorri Agnarsson, snorri@hi.is
2
3  — Þýðið með eftirfarandi skipun:
4  —   ghc -o primes.exe —make primes.hs
5
6  — Notkun: primes
7  — Gildi: Óendanlegur vaxandi listi allra prímtalna,
8  —       [2,3,5,7,11,13,17,...]
9  primes = [2]++[p|p<-[3,5..],isPrime p]
10
11 — Notkun: factor ps n
12 — Fyrir: n er heiltala, stærri en 1, ps er listi
13 —       prímtalna. Engin prímtala sem ekki er
14 —       í ps gengur upp í n.
15 — Gildi: Listi prímpátta n í vaxandi röð, eins oft
16 —       og þeir koma fyrir. Margfeldi talnanna í
17 —       listanum er því n og allar tölurnar eru
18 —       prímtölur.
19 factor (p:ps) n
20     | p*p > n      = [n]
21     | mod n p == 0 = [p] ++ factor (p:ps) (div n p)
22     | True         = factor ps n
23

```



```

24 — Notkun: isPrime n
25 — Fyrir: n er heiltala >= 2.
26 — Gildi: True ef n er prímtala, False annars.
27 isPrime n = (head (factor primes n))==n
28
29 — Skrifum lista 100 fyrstu prímtalnanna
30 main :: IO ()
31 main = do { print (take 100 primes) }

```

## 4.2 Pýþagórasarprenndir

Listi allra Pýþagórasarprennda  $(x, y, z)$  þar sem  $x, y$  og  $z$  eru heiltölur þ.a.  $x^2 + y^2 = z^2$ .

```

1 — Höfundur: Snorri Agnarsson, snorri@hi.is
2
3 — Þýðið með eftirfarandi skipun:
4 — ghc -o Pyth.exe --make Pyth.hs
5
6 pyth1 =
7   [(x,y,z) | y <- [2..],
8               x <- [1..(y-1)],
9               let z=floor $ sqrt $ fromIntegral (x^2+y^2),
10              x^2+y^2==z^2,
11              (gcd (gcd x y) z)==1
12   ]
13
14 pyth2 =
15   do { y <- [2..]
16       ; x <- [1..(y-1)]
17       ; let z=floor $ sqrt $ fromIntegral (x^2+y^2)
18       ; zz <- if x^2+y^2==z^2 then [z] else []
19       ; if (gcd (gcd x y) zz)==1 then [(x,y,zz)] else []
20   }
21
22
23
24 pyth3 =
25   [2..] >>= \y ->
26   [1..(y-1)] >>= \x ->
27   (let
28     z=floor $ sqrt $ fromIntegral (x^2+y^2)
29     in
30     if x^2+y^2==z^2
31     then
32       [z]
33     else
34       []
35   ) >>= \z ->
36   if (gcd (gcd x y) z)==1

```

```
37   then
38     [(x,y,z)]
39   else
40     []
41
42 main :: IO ()
43 main =
44   do { print (take 100 pyth1) }
```

# TÖL304G

## Forritunarmál

### Vikublað 11

Snorri Agnarsson

2. nóvember 2015

## Efnisyfirlit

1	Efni vikunnar	1
2	Skráning verkefnaskila	1

## 1 Efni vikunnar

Við byrjum á að ræða *ruslasöfnun*, sem við komumst ekki yfir í síðustu viku.

Í Uglunni má finna bækling um ruslasöfnun sem þið skuluð lesa.

Í framhaldi af því höldum við áfram að ræða um forritunarmálið Haskell og þið skuluð kíkja á ritlinginn um einstæður (monads) í Haskell.

## 2 Skráning verkefnaskila

Bókhaldið yfir verkefnaskil er eilítið flókið fyrir dæmakennara og komið hefur í ljós að sá hluti þess sem varðar hópverkefni þarfnast endurtalningar og endurskráningar. Allar upplýsingar eiga hins vegar að vera til staðar í Gradescope þannig að þetta er ekki neitt grundvallarvandamál, en það mun krefjast tíma að vinna þessa endurskoðun.

Öll verkefni í námskeiðinu eiga að vera skráð í Gradescope. Ef verkefni er ekki skráð í Gradescope þá er það vegna mistaka og þarfnast þá leiðréttingar. Öll einstaklingsverkefni eiga að vera skráð í Gradescope undir viðkomandi nemanda. Öll hópverkefni eru skráð í Gradescope undir einhverjum þeirra nemenda sem standa að skilunum. Nemendur ættu endilega að kíkja á Gradescope og staðfesta að öll þeirra

skil séu skráð. Ef svo er ekki þá þarf að láta kennara vita af vandamálinu svo hægt sé að reyna að leiðrétta það.

# TÖL304G

## Forritunarmál

### Vikublað 12

Snorri Agnarsson

8. nóvember 2015

## Efnisyfirlit

1	Fjölnota einingar í Jövu	1
2	Fjölnota klasi í Java	2
3	Fjölnota klasar í C++	2

## 1 Fjölnota einingar í Jövu

Í Jövu eru fyrirbæri sem kallast *generics* og *generic types*, sem þýða mætti sem **fjölnota forritun** og **fjölnota tög**. Þau gera okkur kleift að forrita fjölnota einingar og aðferðir.

Lítið á eina<sup>1</sup> eða aðra<sup>2</sup> vefgrein um fjölnota forritun í Java og lesið umfjöllunina um fjölnota tög nægilega vel til að geta leyst skilaverkefnið.

Ég mæli með því að lesa vefsíðuna<sup>3</sup> á Wikipediu um *covariance* og *contravariance*. Þar eru til dæmis athyglisverðar upplýsingar um muninn á virkni Java og C# hvað varðar tögun í fjölnota einingum. Hins vegar munum við í námskeiðinu leggja áhersluna á Java og C++.

Í skilaverkefninu megið þið nota eins mikið úr Java forgangsbiðröðinni í greininni um rökstudda forritun í Java<sup>4</sup> eins og ykkur hentar, eða þið getið farið eigin leiðir.

---

<sup>1</sup><http://docs.oracle.com/javase/tutorial/extra/generics/index.html>

<sup>2</sup><http://docs.oracle.com/javase/tutorial/java/generics/index.html>

<sup>3</sup>[https://en.wikipedia.org/wiki/Covariance\\_and\\_contravariance\\_%28computer\\_science%29](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29)

<sup>4</sup><http://notendur.hi.is/snorri/downloads/rokjava.pdf>

## 2 Fjölnota klasi í Java

Við munum ræða um klasann `MyArray` í fyrirlestri. Forritstextinn fyrir klasann er í ZIP skránni `klasar.zip` í Uglunni.

## 3 Fjölnota klasar í C++

Við munum ræða um klasana `queuearray` og `queuechain` í fyrirlestri. Forritstextinn fyrir klasana eru í ZIP skránni `klasar.zip` í Uglunni, ásamt forritstexta fyrir grunnklasa og prófunarforrit.

# TÖL304G

## Forritunarmál

### Vikublað 13

Snorri Agnarsson

16. nóvember 2015

## Efnisyfirlit

1	Samskeiða forritun í Java	1
2	Samskeiða forritun í Morpho	2

## 1 Samskeiða forritun í Java

Frumstæðasta aðferðin til að gera samskeiða forritun í Java felst í beinni notkun á klasanum Thread. Hér er einfalt Java forrit með þræði.

```
public class ThreadTest
{
    public static void main( String[] args )
    {
        Thread t =
            new Thread()
            {
                public void run()
                {
                    try
                    {
                        for( int i=0 ; i!=10 ; i++ )
                        {
                            Thread.sleep(10000);
                        }
                    }
                }
            }
    }
}
```

```

        System.out.println(i);
    }
}
catch( InterruptedException e )
{
    e.printStackTrace();
}
}
};
t.start();
}
}
}

```

## 2 Samskeiða forritun í Morpho

Notum eftirfarandi Morpho forritstexta

```

rec fun go(@b)
{
    startTask(fun() {b});
};

```

Þar sem skilgreint er fall `go` sem ræsir samskeiða vinnslu sem keyrir sem nýtt *task*. Annar möguleiki er að ræsa samskeiða vinnslu sem nýjan *fiber*, svona:

```

rec fun go(@b)
{
    startFiber(fun() {b});
};

```

Við munum fíkta í þessu og í Java forritinu að ofan í fyrirlestri.