

TÖL304G

Forritunarmál

Vikublað 1

Snorri Agnarsson

25. ágúst 2019

Efnisyfirlit

1	Inngangur	2
2	Hnit kennara	2
3	Lágmörk á verkefnaskilum	3
4	Miðmísserispróf	3
5	Dæmaskil og einkunnir	3
6	Gradescope	4
6.1	Sýnilausnir	4
7	Piazza	4
8	Upptökur fyrirlestra	5
9	Hæfnisviðmið	5
10	Helstu forritunarmál	5
11	Áhersluatriði	6
12	Bækur og annað lesefni	6

13 Áætlun	7
14 Efni vikunnar	7
15 Hvað er „mál“?	7
16 Samhengisfrjáls mál og BNF	7
17 Útleiðslur	9
18 Regluleg mál	10
19 Reglulegar segðir	10
20 Endanlegar stöðuvélar	11
21 Extended BNF	12
22 Dæmatímar og dæmahópar	13

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 mismuni 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. Ef þið lendið í vandræðum með þessar reglur þá ættuð þið að hafa samband við mig, helst áður en ég skrái ykkur úr námskeiðinu.

Í þessu námskeiði þarf að skila að minnsta kosti fjórum einstaklingsverkefnum (Gradescope), fjórum hópverkefnum (Gradescope¹) og fjórum útgöngusvörum úr dæmatímum fyrir lok áttundu viku námskeiðsins. Í flestum vikum, en ekki öllum, verður möguleiki á að skila einu einstaklingsverkefni, einu hópverkefni og einu útgöngusvari. Útgöngusvör eru örstutt svör á stuttum spurningum sem verður skilað bréflaga í lok dæmatíma. Ekki er gefinn einkunn fyrir útgöngusvör.

4 Miðmisserispróf

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

5 Dæmaskil og einkunnir

Í námskeiðinu verða oftast tvenn vikuleg verkefni, eitt einstaklingsverkefni og eitt hópverkefni. Hópverkefni má vinna í dæmatíma. Engin takmörk 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. Hópaskipting má vel vera mismunandi í mismunandi vikum og þið ráðið sjálf hópaskiptingu.

Hópverkefnum og einstaklingsverkefnum skal skila fyrir miðnætti á fimmtudegi í skilaviku verkefnanna. Engum einstaklingsverkefnum eða hópverkefnum skal skila í fyrstu viku.

Fyrir hópverkefni mun einhver í hópnum þurfa að skila verkefninu í Gradescope og sá eða sú mun þá þurfa að tilgreina hverjir aðrir nemendur eru í hópnum.

Meðaltal 7 bestu einkunna einstaklingsverkefna mynda 20% af einkunn námskeiðsins.

Meðaltal 7 bestu einkunna hópverkefna mynda 20% af einkunn námskeiðsins.

Útgöngusvör hafa ekki áhrif á lokaekinn námskeiðsins.

¹<http://gradescope.com>

Útgöngusurning fyrir dæmatíma verður e.t.v. „hver er upphaldsliturinn þinn?“ en svarinu þarf að skila á pappír í dæmatímanum og merkja með nafni og háskólatölvu-póstfangi.

Ef nemendur af einhverjum ástæðum telja sig almennt ekki hafa tök á að skila útgöngusvörum ættu þeir að hafa samband við mig² og sækja um undanþágu.

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

Einkunn úr lokaprófi gildir 60% af lokaekinnun, en athugið að það þarf að ná prófi. Ef prófseinkunnin er falleinkunn munu verkefniseinkunnir verða hunsaðar og prófseinkunnin verður lokaekinnun. Einkunn úr miðmísserisprófi er vegin inn áður en athugað er hvort prófinu hefur verið náð.

Afritun lausna einstaklingsverkefna er forboðin. Þó er í lagi og mælt er með að nemendur ræði saman sín á milli um dæmin og lausnarhugmyndir tengdar þeim. En hver og einn nemandi skal skrifa (og prófa og sýna prófanir, ef hægt er) sínar eigin lausnir og skila þeim.

6 Gradescope

Gradescope kerfið hefur gefið góða raun og við munum nota það í þessu námskeiði bæði fyrir einstaklingsverkefni og hópverkefni.

Lausnum á einstaklingsverkefnum og hópverkefnum skal skila í Gradescope í gegnum vefviðmót Gradescope.

6.1 Sýnilausnir

Fyrir hver heimaæmi verður e.t.v. stundum valin ein lausn nemanda sem sýnilausn. Ef þið viljið ekki að lausnin ykkar komi til greina merkið þá lausnina með „**EKKI SÝNILAUSN**“ efst á fyrstu blaðsíðu.

7 Piazza

Við munum nota spjallkerfið Piazza³ 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.

²mailto:snorri@hi.is

³www.piazza.com

8 Upptökur fyrirlestra

Fyrirhugað er að taka fyrirlestra upp sem myndskaið í Panopto kerfinu sem verða aðgengileg gegnum Ugluna⁴ í sérstökum flipa efst á síðu námskeiðsins. Athugið að nota frekar Ugluna en Moodle til að nálgast upptökurnar. Moodle verður trúlega ekki notað í þessu námskeiði.

9 Hæfnisviðmið

Við lok námskeiðsins mun nemandi

- skilja hvernig málfræði forritunarmála er lýst, geta komist að því hvort tiltekið forrit sé málfræðilega rétt fyrir gefna mállýsingu, geta skrifað mállýsingar fyrir einföld mál
- skilja hvað rökstudd forritun er og geta forritað á rökstuddan hátt
- skilja og geta útskýrt hvernig umdæmi breytu virkar í ýmsum forritunarmálum ásamt því hvernig minnissvæðum og vakningarfærslum fyrir breytur er úthlutað og tengd saman í ýmsum forritunarmálum
- skilja hinar ýmsu gerðir viðfangaflutninga og geta notað þær
- skilja fallsforritun, listavinnslu, endurkvæmni og halaendurkvæmni og geta notað þær
- skilja hvað ruslasöfnun er og geta lýst nokkrum aðferðum fyrir ruslasöfnun
- skilja hvað fjölnota einingar eru og geta forritað fjölnota einingar í nokkrum forritunarmálum

10 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.

Einnig munum við eilítið nota forritunarmálið Dafny⁵, en einungis til að hnykkja á hugmyndum um rökstudda forritun.

⁴<http://ugla.hi.is>

⁵<https://rise4fun.com/Dafny/IxYN5>

11 Áhersluatriði

Áhersla verður lögð á eftirfarandi atriði.

- Málfræði forritunarmála, BNF, EBNF og málrít. 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 fallsforritun, 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).
- 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.
- Líklega 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.

12 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. EBNF⁶
2. Veforðabók⁷

13 Áætlun

Líkleg efnisröð í námskeiðinu er í skjalinu `fmal_plan.pdf` í Uglunni.

14 Efni vikunnar

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

15 Hvað er „mál“?

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

16 Samhengisfrjáls mál og BNF

BNF⁸ er aðferð, svokallað meta-mál, til að skilgreina mál. Með BNF er í grundvallaratriðum 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

⁶https://en.wikipedia.org/wiki/Extended_Backus-Naur_form

⁷<http://foldoc.org>

⁸https://en.wikipedia.org/wiki/Backus-Naur_form

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 og 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
```

eða, betur sniðsett:

```

$$\begin{aligned} \langle expr \rangle &::= \langle num \rangle \\ &| ( \langle expr \rangle ) \\ &| \langle expr \rangle \langle op \rangle \langle expr \rangle \\ \langle op \rangle &::= + \\ &| - \\ &| * \\ &| / \\ \langle num \rangle &::= \langle digit \rangle \\ &| \langle digit \rangle \langle num \rangle \\ \langle digit \rangle &::= 0 \\ &| 1 \\ &| 2 \\ &| 3 \\ &| 4 \\ &| 5 \\ &| 6 \\ &| 7 \\ &| 8 \\ &| 9 \end{aligned}$$

```

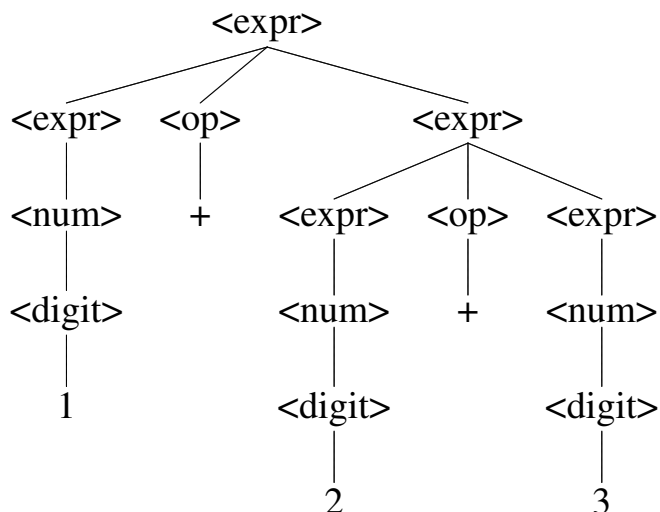
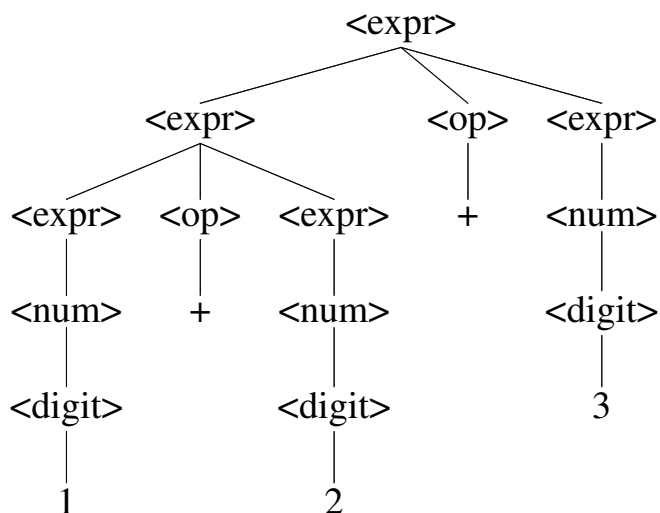
Þessi BNF skilgreining lýsir máli sem inniheldur strengi sem eru segðir („formúlur“) með heiltölugildum og venjulegum reikniaðgerðum.

17 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> =>  
1 + <expr> <op> <expr> =>  
... =>  
1 + 2 + 3
```

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



Ætlast er til að þið getið búið til slík útleiðslutré fyrir gefinn streng í máli sem skilgreint er með tiltekinni BNF skilgreiningu.

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 samhengisfrjáls 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 sjálft, eru EBNF (Extended BNF) og málrít (syntax diagrams). Ætlast er til að þið getið lesið og skilið allar þessar þrjár aðferðir.

18 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 stöðuvélar⁹ (finite state automaton, finite state machine) og reglulegar segðir (regular expressions) til að lýsa einföldum reglulegum málum.

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

19 Reglulegar segðir

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

- \emptyset er regluleg segð sem skilgreinir málið $M(\emptyset) = \emptyset = \{\}$
- ϵ er regluleg segð sem skilgreinir málið $M(\epsilon) = \{\epsilon\}$
- Ef $x \in \Sigma$ þá er x regluleg segð sem skilgreinir málið $M(x) = \{x\}$, sem er eins staks mengi sem inniheldur eins stafs streng.

⁹https://en.wikipedia.org/wiki/Finite-state_machine

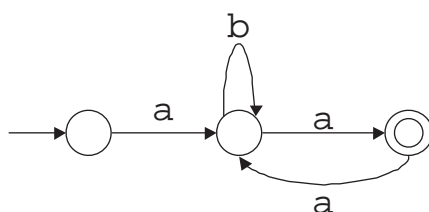
- Ef x og y eru reglulegar segðir þá er xy regluleg segð sem skilgreinir málið $M(xy) = \{uv \mid u \in M(x) \wedge v \in M(y)\}$.
- Ef x og y eru reglulegar segðir þá er $x \mid y$ regluleg segð sem skilgreinir málið $M(x \mid y) = \{w \mid w \in M(x) \vee w \in M(y)\}$.
- Ef x er regluleg segð þá er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

Í ofangreindum skilgreiningum gerum við ráð fyrir að séu u og v eru strengir þá sé strengurinn uv skilgreindur sem samskeyting u og v . Svipað gildir um samskeytingu fleiri en tveggja strengja. 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).

20 Endanlegar stöðuvélar

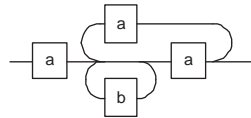
Eins og fram hefur komið eru endanlegar stöðuvélar enn ein aðferð til að skilgreina regluleg mál. Þessi mynd sýnir dæmi um endanlega stöðuvél sem ber kennsl á strengi yfir stafrófið $\{a,b\}$. Í málinu eru m.a. strengirnir aa , aba , $abba$, o.s.frv., en einnig má stinga aa einhvers staðar inn í löglegan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



Takið eftir að í málritinu 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 málrit, og geti borið kennsl á hvort tilteknir strengir séu í málinu sem slíkar vélar og rit skilgreina. Fleiri dæmi um málrit má finna í handbókinni fyrir forritunarmálið Morpho, sem mun verða sett í Ugluna þegar þar að kemur.

21 Extended BNF

Extended BNF¹⁰ 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 staðall¹¹ fyrir EBNF, sem líklegt er að flestir notendur EBNF fari nálægt að fara eftir. Sjá einnig grein eftir R.S. Scowen¹².

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 mál fyrirbærum í hægri hlutum reglna, sem eiga rætur að rekja til reglulegra segða:

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

¹⁰Sjá t.d. bls. 131-134 í Sebesta.

¹¹<http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html>

¹²<http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf>

Samkvæmt staðaltillögunni sem liggur fyrir er samskeyting í EBNF táknuð með aðgerðinni , (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' } ;
```

22 Dæmatímar og dæmahópar

Dæmatímar munu hefjast í viku 2.

Þið ráðið því sjálf í hvaða dæmatíma þið mætið í hverri viku. Ef það veldur því að of margir hrúgast saman þá munum við reyna að taka á því.

TÖL304G

Forritunarmál

Vikublað 2

Snorri Agnarsson

1. september 2019

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 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 kynnst Morpho betur síðar. Sækja má Morpho á vefnum¹, en það er varla tímabært ennþá.

¹<http://morpho.cs.hi.is>

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 vefnum². 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 vefnum³ 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:

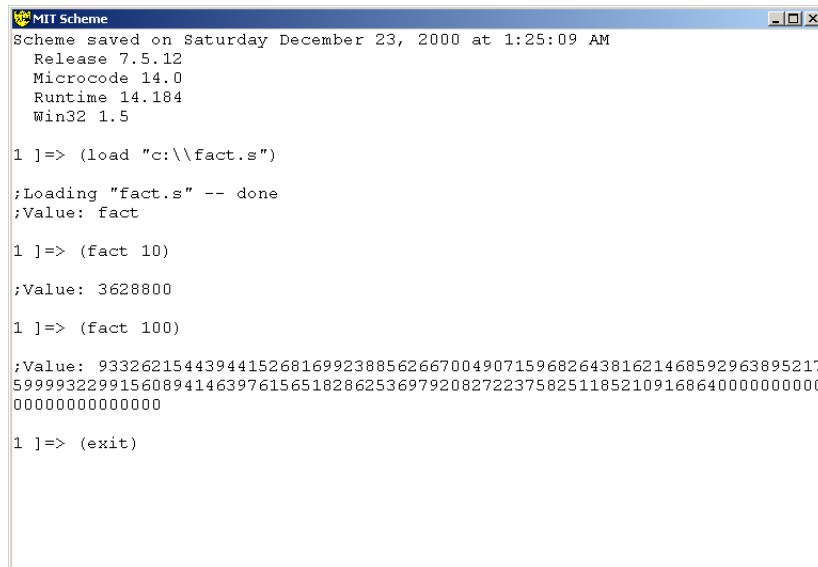
1. Ræsum Scheme úr valblaðsliðnum `Start → Programs → MIT Scheme → Scheme`.
2. Notum ritil til að skrifa eða breyta forritinu (skránni) `C:\Users\Nonni\TÖL304\fact.s`. (Væntanlega ekki Nonni, samt.)
3. Hlöðum forritinu í Scheme með skipuninni

```
(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.

²<http://racket-lang.org/>

³<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, 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 vefnum⁴ 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 skrifuð 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)))))
```

⁴<https://rise4fun.com/Dafny/xR7n>

```

        (cons (car x) (append1 (cdr x) y))
    )
)
;; stofn fallisins 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)))
    )
  )
)
```

)
hjalp
)

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

TÖL304G

Forritunarmál

Vikublað 3

Snorri Agnarsson

8. september 2019

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 λ -formúlur (lambda formúlur), og reikninga með slíkar formúlur.

λ -formúlur eru skilgreindar á eftirfarandi hátt:

- Ef x er breytunafn þá er x lögleg λ -formúla.

- Ef x er breytunafn og N er lögleg λ -formúla þá er $\lambda x.N$ lögleg λ -formúla.
- Ef M og N eru löglegar λ -formúlur þá er MN lögleg λ -formúla.
- Ef M er lögleg λ -formúla þá er (M) lögleg λ -formúla.
- Engar aðrar formúlur eru löglegar λ -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 λ -formúla sem samsett er úr minni λ -formúlum M_1 , M_2 og M_3 þá túlkum við hana sem jafngilda λ -formúlunni $(M_1 M_2) M_3$ ¹. Hins vegar er $\lambda x.M_1 M_2$ talin jafngild $\lambda x.(M_1 M_2)$.

4.1 Bindingar

Í λ -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 λ -formúlunni x er frjáls.
- Ef N og M eru λ -formúlur þá eru allar tilvísanir í breytu x frjálsar í (NM) , sem eru frjálsar í N og M .
- Ef N er λ -formúla þá eru engar tilvísanir í breytuna x í λ -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 λ -formúlu sem viðfang og skilar mengi frjálsra breyta í formúlunni:

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

¹Mjög 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 λ -reiking, verðum við að setja svigana nákvæmlega svona.

4.2 Innsetningar

Í λ -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 λ -formúlu og fá út nýja λ -formúlu². Formúlan $\{x \rightarrow M\}N$, þar sem M og N eru λ -formúlur, er ekki sjálf λ -formúla, en stendur fyrir þá λ -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 λ -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 λ -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 λ -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 λ -reikningi eru skilgreindar reiknireglur, sem lýsa því hvaða aðgerðir má gera á λ -formúlur án þess að breyta „gildi“ þeirra. Reglurnar eru eftirfarandi:

- (β -jafngildi) Ef N og M eru λ -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 λ -reikningi að útkoman verður sú sama hvor leiðin sem farin er. Til dæmis getum við í λ -reikningi³ skrifað bæði

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

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

³Með örlitlum viðbótum við hreinan λ -reikning, til að leyfa aðeins flóknari formúlur, eins og einnig er gert í ýmsri umfjöllun um λ -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 λ -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

15. september 2019

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 sjálfsgöðu, 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 λ -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 vefnum¹ má finna skjal um „óendanlega“ strauma í Scheme, ásamt Scheme forritstexta² fyrir föllin þar.

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

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

TÖL304G

Forritunarmál

Vikublað 5

Snorri Agnarsson

22. september 2019

Efnisyfirlit

1	Efni vikunnar	1
2	Fikt með lokanir í Scheme	2
2.1	xcons, xcar og xcdr	2
2.2	Straumar	3
3	Rökstudd forritun	4

1 Efni vikunnar

Við höldum áfram með Scheme, lokanir, bálkmótun og strauma. 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 stýrihlekkir (*control link*, *dynamic link*) annars vegar, og gegnum tengihlekkir (aðgangshlekkir, *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, eins og minnst hefur verið á.

2 Fikt með lokanir í Scheme

2.1 xcons, xcar og xcdr

Íhugum eftirfarandi í Scheme.

```
;; Notkun: (xcons x y)
;; Fyrir: x og y eru hvaða gildi sem er.
;; Gildi: Fall c með eftirfarandi lýsingu:
;;         Notkun: (c z)
;;         Fyrir: z er satt eða ósatt (#t eða #f).
;;         Gildi: x ef z er satt, y annars.
(define (xcons x y) (lambda (z) (if z x y)))

;; Notkun (xcar (xcons x y))
;; Fyrir: x og y eru hvaða gildi sem er.
;; Gildi: x.
(define (xcar c) (c #t))

;; Notkun (xcdr (xcons x y))
;; Fyrir: x og y eru hvaða gildi sem er.
;; Gildi: y.
(define (xcdr c) (c #f))
```

Við sjáum að virkni þessa falla er svipuð virkni innbyggðu fallanna `cons`, `car` og `cdr`. Lykilatriði hér er að ljóst er að fallið `xcons` pakkar greinilega saman tveimur gildum í eitt, og föllin `xcar` og `xcdr` sækja þau gildi. Það eru sömu vensl milli fallanna `xcons`, `xcar` og `xcdr` og eru milli fallanna `cons`, `car` og `cdr`.

```
(xcar (xcons x y)) = x
(xcdr (xcons x y)) = y
```

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

Jafnaðarmerkið þýðir að segðin vinstra megin skilar sama gildi og segðin hægra megin. `x` og `y` geta verið hvaða gildi sem er. Ef hliðarverkanir eru ekki til staðar (eins og við tryggjum) þá mega `x` og `y` vera hvaða segðir sem er.

Við sjáum því að föllin `cons`, `car` og `cdr` eru í raun óþarfi sem grunnföll því við gætum búið til þeirra virkni út frá annarri grunnvirkni í Scheme. Hins vegar er hraðvirkara að nota þau og auk þess fáum við að nota fallið `pair?` sem segir til um hvort eitthvert tiltekið gildi er par.

Athugið að ef við hefðum ekki þann möguleika í Scheme að skila lokun sem gildi falls þá væri þetta ekki hægt.

2.2 Straumar

Straumar í Scheme eru skemmtileg fyrirbæri sem við munum líta á. Sumar útgáfur Scheme hafa innbyggða virkni fyrir strauma en í öðrum útgáfum er auðvelt að útfæra þá. Straumar, svipað og listar, byggja á samspili þriggja aðgerða sem hafa eftirfarandi eiginleika:

```
(stream-car (cons-stream x y)) = x  
(stream-cdr (cons-stream x y)) = y
```

Jafnaðarmerkið þýðir, eins og lengra að ofan, að segðin vinstra megin skilar sama gildi og segðin hægra megin. x og y geta verið hvaða segðir sem er. Munurinn á virkninni á `cons`, `car` og `cdr` annars vegar, og `cons-stream`, `stream-car` og `stream-cdr` hins vegar, er að `cons-stream` er ekki fall, heldur lykilorð, og seinna viðfangið í `cons-stream` er ekki reiknað fyrr en fallinu `stream-cdr` er beitt á útkomuna úr segðinni `(cons-stream x y)`. Þetta veldur því að hægt er að reikna skilgreiningar eins og þessa:

```
(define e (cons-stream 1 e))
```

Þetta býr til óendanlegan straum af 1 því segðin `(stream-car e)` skilar 1, og segðin `(stream-car (stream-cdr e))` skilar einnig 1 vegna þess að segðin `(stream-cdr e)` skilar `e`, og svo koll af kolli.

Á vefnum¹ má finna skjal um „óendanlega“ strauma í Scheme, ásamt Scheme forritstexta² fyrir föllin þar.

Útfæra má strauma á ýmsan hátt, en þægilegt er að gera það þannig að eftirfarandi jafngildi séu notuð:

```
(cons-stream x y) = (cons x (delay y))  
(stream-car z) = (car z)  
(stream-cdr z) = (force (cdr z))
```

Lykilorðið `delay` virkar þannig að segðin `(delay e)`, þar sem e er hvaða segð sem er, skilar svokölluðu *loforði* (promise), sem inniheldur segðina e , án þess að segðin hafi enn verið gilduð (evaluated). Sé fallinu `force` seinna beitt á þetta loforð þá verður segðin e gilduð og gildinu skilað. Við höfum því þetta jafngildi:

```
(force (delay e)) = e
```

Sé fallinu `force` beitt aftur er skilað sama gildi án þess að það þurfi að endurreikna það. Við munum seinna sjá svipaða virkni í forritunarmálinu Haskell, en í því tilviki er það hluti af grundvallarvirkni forritunarmálsins og krefst ekki beitingu neinna sérstakra aðgerða.

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

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

Í framhaldinu má auðveldlega útfæra föllin `stream-car` og `stream-cdr` svona:

```
(define stream-car car)
(define (stream-cdr z) (force (cdr z)))
```

Hins vegar þarf aðrar aðferðir til að útfæra `cons-stream` því það er ekki fall heldur lykilorð, eins og `delay`. Allar útfærslur Scheme hafa einhverjar aðferðir til að útfæra ný lykilorð, en við munum ekki fjalla um slíkt, eða a.m.k. er ekkert slíkt til prófs.

3 Röktudd forritun

Finna má á vefnum³ skjal um röksemdafærsluaðferðir í forritun. Þar eru meðal annars dæmi um notkun á ýmsum gerðum stöðulýsinga sem mikilvægar eru í röktuddri 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 og einingaforritun, þá verður krafist fastayrðingar gagna (data invariant) fyrir sérhvert nýtt gagnamót (data structure) sem skilgreint er.

³<http://www.hi.is/~snorri/downloads/rokjava.pdf>

TÖL304G

Forritunarmál

Vikublað 6

Snorri Agnarsson

29. september 2019

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 (*typing*) í þeim. Þau eru rammtöguð (*strongly typed*) 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 tómi listinn. Ef til dæmis `'a` og `'b` eru tvö tög þá er `'a * 'b` tag para 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 INRIA¹. 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 vefnum².

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
*)

```

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

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

```

let hd x =
  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,  $f: A \rightarrow B \rightarrow B$ ,
       u er gildi af tagi B,
        $x=[x_1; \dots; x_N]$  er listi gilda af
       tagi A.
Gildi:  $x_1+x_2+\dots+x_N+u$ , 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 =  $[x_1; x_2; \dots; x_N]$ , f er 'a -> 'b
** Eftir: y er listinn  $[f x_1; f x_2; \dots; f x_N]$ 
*)
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

7. október 2019

Efnisyfirlit

1	Miðmisserispróf	1
2	Morpho	1
2.1	Morpho keyrsluumhverfið	3
3	Notkun Morpho	5

1 Miðmisserispróf

Miðmisserisprófið verður haldið fimmtudaginn 17. október í fyrirlestratímunum í N-132 kl. 10:00-11:30. Engin hjálpargögn verða leyfð í prófinu.

2 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 (morpho.jar) og handbókina (Morpho.pdf) má finna í

Uglunni.

2.1 Morpho keyrsluumhverfið

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

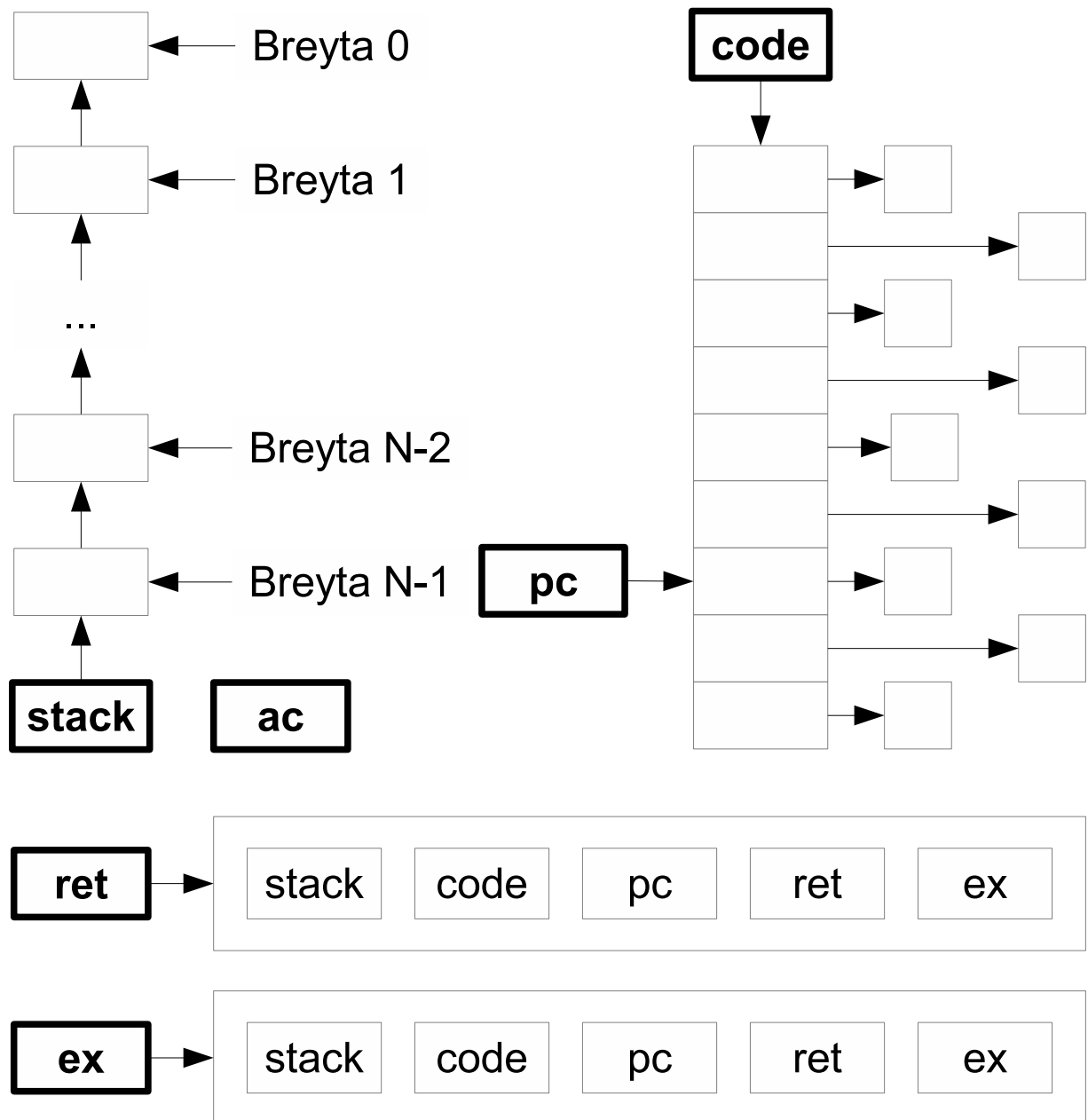
ret. Eðlilegt framhald úr núverandi falli (sjá neðar).

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 hlekk í gildahlaðanum. Þetta veldur minnisparnaði þegar breytur þurfa að lifa það af að fallið deyr sem bjó þær til, en þið þurfið



Mynd 1: Staða í keyrslu Morpho

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

Hitt framhaldið er í gistinu **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.

3 Notkun Morpho

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

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

Til að þýða Morphoforrit úr textaskrá `x.morpho` má nota eftirfarandi skipun:

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

Út úr slíkri þýðingu gæti, til dæmis, komið út keyrsluhæf skrá `x.mexe`, sem þá má keyra með skipuninni

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

Nánari upplýsingar er að finna í handbókinni fyrir Morpho.

TÖL304G

Forritunarmál

Vikublað 8

Snorri Agnarsson

13. október 2019

Efnisyfirlit

1	Miðmisserispróf	1
2	Upplýsingahuld og hönnunarskjöl	3

1 Miðmisserispróf

Fimmtudaginn 17. október verður miðmisserispróf. Einkunnin í miðmisserisprófinu gildir til hækkunar á prófseinkunn í námskeiðinu, ef einkunnin er hærri en einkunnin á lokaprófi. Í því tilfelli gildir miðmisserisprófseinkunnin 30% á móti lokaprófseinkunninni.

Miðmisserisprófið mun einnig gilda sem ein verkefnaskil. Skilafrestur fyrir verkefni 7 verður lengdur fram í næstu viku, til fimmtudagsins 24. október. Auk þess að taka miðmisserisprófið á fimmtudaginn verður einnig unnt að skila prófinu sem hópverkefni í Gradescope. Prófið verður sett í Ugluna á fimmtudaginn og skilafrestur til að skila prófinu sem verkefni verður til kl. 13:20 mánudaginn 21. október. Próf sem tekið er sem verkefni verður skilgreint í Gradescope sem hópverkefni og engin takmörk eru á fjölda þeirra nemenda sem mega vinna saman í að leysa prófið á þann hátt. Einnig má þá nota öll hjálpargögn, en prófið á fimmtudaginn er hins vegar algerlega án hjálpargagna.

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

- Mállýsingar

- BNF, EBNF og málrit
- Regluleg mál, reglulegar segðir, endanlegar stöðuvélar, löggengar og brigðgengar.
- Viðfangaflytningar
 - Gildisviðföng (call by value)
 - Tilvísunarviðföng (call by reference)
 - Nafnviðföng (call by name)
 - Löt viðföng (lazy evaluation)
- Bálmótun og vakningarfærslur (*block structure, activation records*)
 - Innihald vakningarfærslu (*activation record*) með og án bálmótunar
 - * Viðföng
 - * Staðværar breytur
 - * Tengihlekkur (aðeins í bálmótuðum)
 - * Vendivistfang
 - * Stýrihlekkur

Viðföng, staðværar breytur og tengihlekkur mynda **umhverfi** (*environment*), og vendivistfang ásamt stýrihlekk mynda **framhald** (*continuation*).
 - Tengihlekkir (aðgangshlekkir) (*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*)
- Rökstudd forritun
 - Notkunarlýsingar: „Notkun:“, „Fyrir:“ og „Eftir:“
 - Fastayrðing lykkju
- Tögun án breytuyfirlýsinga (CAML)
 - Ætlast er til að þið getið greint tag einfaldra falla í CAML.
- Scheme
 - Bálmótað fallsforritunarmál með keyrslutögun
 - Afbrigði af LISP
- CAML
 - Bálmótað fallsforritunarmál með rammtögun
 - Tagað án breytuyfirlýsinga
- Morpho
 - Bálmótað einingaforritunarmál með listavinnslu

Prófið verður án hjálpargagna og athugið að líta skal á öll heimaverkefni sem efni til prófs, nema skilafrestur viðkomandi verkefna sé ekki útrunninn.

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 oftar 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 9

Snorri Agnarsson

20. október 2019

Efnisyfirlit

1	Hlutbundin forritun í Morpho	1
1.1	Einfaldur hlutbundinn hlaði	1
1.2	Hlaði með víxlunarboði	3
1.3	Hlaðaeining með innfluttum arfi	4
1.4	Samband boða og aðferða	6
1.5	Hlaði með uppnefndum boðnöfnum	7

1 Hlutbundin forritun í Morpho

1.1 Einfaldur hlutbundinn hlaði

Dæmi um klasaskilgreiningu í Morpho er eftirfarandi eining. Athugið að einstaklingsverkefni núverandi viku er einnig hlaði, en sá hlaði má ekki vera hlutbundinn þannig að ekki dugar að skila þessari einingu sem lausn.

```
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 s.
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 fjarlægt 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.

1.2 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"
36 ;

```

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 lykilorðini (í þessu dæmi kallið `stack()`) verður að skila hlut.

1.3 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" ;

```

1.4 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 yfirklassa. 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.

1.5 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 "hladi.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 "hladaprof.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 2019

Efnisyfirlit

1 Efni vikunnar	1
2 Haskell	1
3 Náms efni í Haskell	1
3.1 Vefgæði	2
3.2 Löt gildun	2
3.3 Listaritháttur	3
4 Haskell dæmi	5
4.1 Prímtölur	5
4.2 Pýþagórasarprenndir	5

1 Efni vikunnar

Við höldum áfram með Morpho, förum síðan í Haskell forritunarmálið og ræðum einnig *ruslasöfnun* þegar tími gefst til.

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 vefnum¹ og sækja má Haskell þýðendur þaðan. Við munum nota Glasgow Haskell².

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!³.

Einnig má finna vefsíðuna Try Haskell⁴ 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:

¹<http://www.haskell.org>

²<http://www.haskell.org/platform/>

³<http://learnyouahaskell.com>

⁴<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** $(\lambda i \rightarrow i^2)$ 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 $z = \text{floor } \$ \text{sqrt } \$ \text{fromIntegral } (x^2+y^2)$

- Síur (*filters*) svo sem

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

Á Haskell síðunni⁵ 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:

⁵<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 vaxandi
13 —       listi ójafnra prímtalna. Engin prímtala
14 —       sem ekki er 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 pyth3 =
23   [2..] >>= \y ->
24   [1..(y-1)] >>= \x ->
25   (let
26     z=floor $ sqrt $ fromIntegral (x^2+y^2)
27     in
28     if x^2+y^2==z^2
29     then
30       [z]
31     else
32       []
33   ) >>= \z ->
34   if (gcd (gcd x y) z)==1
35   then
36     [(x,y,z)]

```

```

37     else
38         []
39
40     pyth4 =
41         concatMap
42             (\y ->
43                 (filter
44                     (\(x,y,z) -> (gcd (gcd x y) z)==1)
45                     (filter
46                         (\(x,y,z) -> x^2+y^2==z^2)
47                         (concatMap
48                             (\x ->
49                                 let
50                                     z=floor (sqrt (fromIntegral (x^2+y^2)))
51                                 in
52                                     [(x,y,z)]
53                             )
54                             [1..(y-1)]
55                         )
56                     )
57                 )
58             )
59         [2..]
60
61     pyth5 =
62         filter
63             (\(x,y,z) -> (gcd (gcd x y) z)==1)
64         (filter
65             (\(x,y,z) -> x^2+y^2==z^2)
66             (concatMap
67                 (\y ->
68                     (
69                         concatMap
70                             (\x ->
71                                 let
72                                     z=floor (sqrt (fromIntegral (x^2+y^2)))
73                                 in
74                                     [(x,y,z)]
75                             )
76                             [1..(y-1)]
77                     )
78                 )
79             [2..]
80         )
81     )
82
83     main :: IO ()
84     main =
85         do
86             print (take 10 pyth1)

```

```
87      print (take 10 pyth2)
88      print (take 10 pyth3)
89      print (take 10 pyth4)
90      print (take 10 pyth5)
```

TÖL304G

Forritunarmál

Vikublað 11

Snorri Agnarsson

3. nóvember 2019

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.

TÖL304G

Forritunarmál

Vikublað 12

Snorri Agnarsson

10. nóvember 2019

Efnisyfirlit

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

1 Fjölnota einingar í Java

Í Java 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 eina¹ eða aðra² 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 vefsíðuna³ 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++.

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

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

³https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

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

17. nóvember 2019

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.

TÖL304G

Forritunarmál

Vikublað 14

Snorri Agnarsson

20. nóvember 2019

Efnisyfirlit

1	Engin hjálpargögn í prófi	1
2	Áherslupunktur fyrir próf	1

1 Engin hjálpargögn í prófi

Engin hjálpargögn verða leyfð í prófinu.

2 Áherslupunktur fyrir próf

Eftirfarandi lista má hafa til hliðsjónar þegar þið eruð að rifja upp námsefnið fyrir próf.

- **Verkefni og vikublöð.** Öll heimaverkefni eru að sjálfsögðu til prófs og ráðlegt er að skilja vel hvernig á að leysa þau öll. Það er öruggt að hluti prófsins mun byggja á heimaverkefnum. Vikublöð skal lesa vel, en ekki er alveg eins mikilvægt að lesa önnur rit sem vísað er í á vefnum, nema sem bakgrunns upplýsingar eða til nánari upplýsingar.
- **Einingaforritun, sérstaklega fjölnota einingar (Morpho og Java).** Ætlast er til að þið getið lesið og skilið forritstexta fyrir fjölnota einingar í þessum forritunarmálum, og að þið getið skrifað fjölnota einingar í þeim.
 - Ytri skjölun eininga, forskilyrði og eftirskilyrði.

- Innri skjölun eininga, fastayrðingar gagna (innri hönnun).
 - Hlutverk í einingaforritun (hönnun, notkun, smíð).
 - Skynsamleg ytri hönnun, þ.e. lýsingar falla, boða og annarra innfluttra og útfluttra atriða.
 - Upplýsingahuld og tilgangur hennar.
 - Rökrænt samband hönnunar, smíðar, notkunarlýsinga og fastayrðingar gagna.
 - Samband forskilyrða og eftirsilyrða í yfirklassa og undirklassa.
- **Listavinnsla og fallsforritun (Scheme, Morpho, CAML og Haskell).**
 - Sígild algrím í listavinnslu, s.s. snúningur, afritun, skeyting, beiting falls á lista, innsetning tvíundaraðgerða á ýmsan hátt.
 - Straumar og sú lata gildun sem tengist þeim.
 - Notkun halaendurkvæmni, meðhöndlun halaendurkvæmni í Scheme og Morpho.
 - Sérstök atriði í Haskell eru löt gildun (*lazy evaluation, call by need*), listarithátturinn (*list comprehension*) ásamt almennu hugmyndinni um einstæður.
- **Innviðir bálmótaðra forritunarmála (Scheme, CAML, Morpho, Haskell).**
 - Innihald vakningarfærslna s.s. tengihlekkir (aðgangshlekkir, *access link, static link*), stýrihlekkir (*control link, dynamic link*), vendivistfang (*return pointer*).
 - Tenging ruslasöfnunar og öflugrar bálmótunar. Mismunandi notkunar-möguleikar fyrir lokanir í Scheme, CAML, Morpho, Haskell annars vegar og hins vegar bálmótaðra forritunarmála þar sem vakningarfærslurnar eru á hlaða, s.s. Pascal.
 - Aðgangshlekkir (tengihlekkir, *access link, static link*) og stýrihlekkir (*control link, dynamic link*)
 - Lokanir (*closures*)
 - Skilgreiningar nafna, bindingar og umdæmi skilgreiningar (*scope*), földun (*nesting*), földunarhæð (*nesting level*)
 - Frjáls breyta (*free variable*), bundin breyta (*bound variable*)
- **Mállýsingar.**
 - BNF, EBNF og málrit.
 - Regluleg mál, reglulegar segðir, endanlegar stöðuvélar.

- Viðfangaflytningar.
 - Gildisviðföng.
 - Tilvísunarviðföng.
 - Löt viðföng.
 - Nafnviðföng.
- Hlutbundnin forritun (Morpho og Java).
 - Boð og aðferðir og munurinn á þeim og sambandið milli þeirra.
 - Skjölun boða, og þá einnig sérstaklega með tilliti til erfða.
- Ruslasöfnunaraferðir (Morpho, Scheme, CAML, Java, Haskell).
 - Merkja og sópa (*mark and sweep*).
 - Tilvísunartalning (*reference count*).
 - Afritunarsöfnun (*stop and copy*).
- Tögun í CAML og Haskell: Tög sem innihalda frjálsar tagbreytur. Þið þurfið að geta fundið út tagið á einföldum CAML eða Haskell skilgreiningum.
- Fallsforritun (*functional programming*)
 - Forritun „án hliðarverkana“ í Haskell, Scheme og CAML
 - Endurkvæmni og halaendurkvæmni og kostir halaendurkvæmni
 - Æðra stigs föll: Föll sem skila föllum, taka föll sem viðföng
 - Listavinnsla (*list processing*)
 - Haskell atriði: Löt gildun, listaritháttur, do-ritháttur, Haskell einstæður: Listar, Maybe.