Markmið í þýðandaverkefni

Snorri Agnarsson

8. janúar 2018

Efnisyfirlit

1 Inngangur

Úr þýðandaverkefninu koma eftirfarandi afurðir sem einkunn verkefnisins verður byggð á:

1. Þýðandi sem les forritstexta í Lisp, Morpho eða svipuðu máli og skilar forritstexta í Morpho-smalamáli. Þessi afurð skiptist í eftirfarandi:
   a) Keyrsluhæfur þýðandi (exe-skrá, jar-skrá eða svipað).
   b) Forritstexti og annað (s.s. makefile) sem til þarf til að smíða þýðandann.
2. Handbók sem lýsir því hvernig setja skal upp þýðandann og nota hann og lýsir forritunarmálinu.

Við munum seinna ræða um handbókina, en á þessu stigi málsins er eðlilegt að byrja strax að kafa í skilgreiningu á virkni þýðandans og skilgreina í stórum dráttum hvernig forritstextarnir sem lesnir eru og skrifaðir af þýðandanum líta út. Við lítum þá í bili á þýðandann sem svartan kassa.

Vonandi hafa flestir í þessu námskeiði notað Morpho. Þeir sem ekki hafa notað Morpho ættu að kynna sér það til að skilja betur hvernig þeirra eigin þýðendur ættu að virka í þessu námskeiði. Þýðendur fyrir Morpho eru sem betur fer einfaldari en flestir aðrir þýðendur, en ástæður þess eru eftirfarandi:

- Málfræði Morpho er einföld.
- Fyrir Morpho er þegar til keyrsluumhverfi (í Java) sem sér um minnimeðhöndlun og hefur þær aðgerðir sem nauðsynlegar eru til að forrit geti haft samskipti við umhverfi sitt.
- Gagnaskipan í Morpho er mjög einföld.
- Í Morpho má þýða hvert stef án tillits til annara stefja eða annars samhengis.

2 Þýðing stefja í Morpho

Hvert stef í Morpho er þýtt fyrir sig, án tillits til annara stefja eða annars samhengis. Úr þýðingunni kemur smalamálsþula sem seinna verður að keyranlegri þulu í minni tölvunnar. Almennt, í nokkurn veginn hvaða forritunarmáli sem er, skiptist útkoman úr þýðingu hvers stefs í þrjá meginhluta: Formála, eftirmála og stofn. Formálinn inniheldur þær skipanir sem framkvæmdar eru í byrjun hverrar inningar (keyrslu) stefsins, eftirmálinn þær skipanir sem framkvæmdar eru í lok hverrar inningar og stofninn þær skipanir sem samsvara þeim setningum eða segðum sem stefið inniheldur. Tilgangur formálans er að ljúka við að smíða vakningarfærslu stefsins (sá sem kallar byrjar smíði vakningarfærslunnar). Tilgangur eftirmálans er að hefja niðurrif vakningarfærslunnar og annarar vinnu sem inna þarf af hendi til að stefið sem kallaði eða stefið sem tekur við af viðkomandi stefi geti haldið áfram sinni keyrslu (stefið sem kallar klárar þessa frágangsvinnu eftir því sem þörf krefur). Í Morpho notum við sérhannað „vélarmál“ sem aftur veldur því að formálar og eftirmálar eru varla til staðar, en við viljum samt skilja að þetta er almenna atburðarásin.

2.1 Dæmi um þýðingu stefs

Íhugum eftirfarandi Morpho forrit:

```
" test .mmod" =
{{
f =
  fun(x)
  {
    x+2;
  };
}};
```

Þýðing þess gæti litið svona út:

```
" test .mmod" =
{{
#"f[f1]" =
  [
  (Fetch 0)
  (Push)
  (MakeVal 2)
  (Call #"+[f2]" 2)
  (Return)
  ];
}}
;
```

Eða svona:

```
" test .mmod" =
{{
#"f[f1]" =
  [
  (Fetch 0)
  (MakeValP 2)
  (CallR #"+[f2]" 2)
  ];
}}
;
```

Þetta er auðvitað óskiljanlegt á þessu stigi málsins (kíkið aftur á þetta þegar þið eruð búin að lesa þennan bækling og annað sem til þarf). Til þess að skilja hvað gengur hér á er nauðsynlegt að vita hvernig vakningarfærslur og hlaðinn í Morpho virka, hvernig föll kalla á föll í Morpho, hvernig þessar Morpho „vélarmálsskipanir“ Fetch, Push, MakeVal, Call, Return, CallR og MakeValP virka, og fleira.

Reyndar er hægt að leysa þýðandaverkefnið í þessu námskeiði og nota aðeins eftirfarandi „vélarmálsskipanir“: Go, GoFalse, Return, MakeVal, Call, Push, Store, Fetch. Ráðlegt er samt

að kynna sér og nota fleiri skipanir.

3 Keyrslustöður

Keyrandi Morpho forrit samanstendur af safni af samhliða keyrandi verkum (*tasks*). Hvert verk samanstendur af safni af þráðlingum (*fibers, coroutines*) sem skiptast á að keyra innan verksins. Til að einfalda málið munum við yfirleitt ímynda okkur að aðeins sé um eitt verk að ræða, með einum þráðlingi.

Þráðlingurinn keyrir Morpho vélarmálsskipanir og hver slík skipun veldur því að ástandið breytist. Ástand þráðlingsins samanstendur af eftirfarandi:

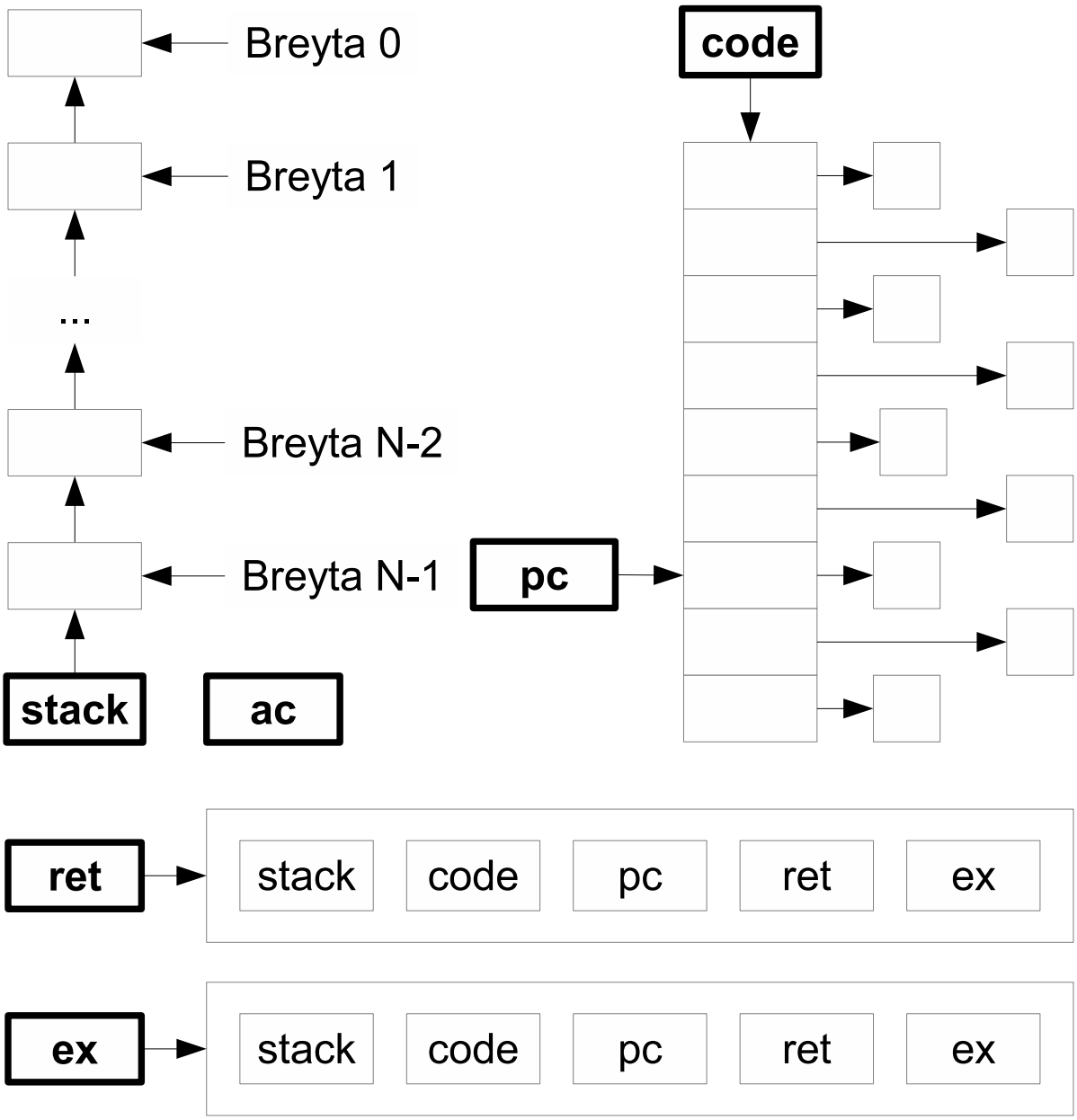
- Skipanafylki (sýndargisti code)
- Staðsetning í skipanafylki (sýndargisti pc)
- Gildahlaði (sýndargisti stack)
- Eðlilegt framhald (sýndargisti ret)
- Afbrigðilegt framhald (sýndargisti ex)
- Skilagildi (sýndargisti ac)

Mynd 1 á næstu síðu sýnir keyrslustöðu þráðlings í Morpho sýndarvél.

Hvert skref í keyrslu þráðlingsins felst í að framkvæma þá skipun í skipanafylkinu (code) sem staðsetningin (pc) vísar á. Oftast er staðsetningin þá um leið færð á næstu skipun á eftir (pc++), en sumar skipanir s.s. Call, Return og Go hafa annars konar áhrif á staðsetninguna. Margar skipanir hafa áhrif á gildahlaðann, sem er notaður til að geyma flest þau gildi sem verið er að vinna með, þ.m.t. allar staðværar breytur (í öllum földunarhæðum) og viðföng. En auk þess er sýndargisti skilagildis (ac) mikið notað undir milliniðurstöður. Gildahlaðinn er ótakmarkaður að stærð, en hann verður samt oftast ekki mjög stór. Á hverju andartaki í keyrslu, fyrir og eftir framkvæmd skipunar, inniheldur gildahlaðinn eftirfarandi upplýsingar, í röð frá toppi hlaðans til botns:

1. Milliniðurstöður í útreikningum núverandi falls, svo sem viðföng fyrir föll sem verið er að undirbúa köll á.
2. Breytur í núverandi földunarhæð, þ.e. staðværar breytur núverandi falls eða bálks og viðföng núverandi falls eða bálks.
3. Breytur í efri földunarhæðum, í röð frá neðstu földunarhæð til efstu.

Athugið að sama breyta getur tekið þátt í mörgum mismunandi gildahlöðum, jafnvel samtímis. Gildahlaðinn er útfærður í Morpho keyrsluumhverfinu sem eintengdur listi. Líta má á hvern hlekk í gildahlaðanum sem breytustaðsetningu. Gildið í breytunni getur breyst en aldrei er gerð breyting á því hver næsti hlekkur er á eftir í keðjunni.



Mynd 1: Staða þráðlings í Morpho sýndarvél með N gildi á hlaða

Þegar kallað er á fall með skipunum svo sem Call eða CallR, er smíðaður nýr gildahlaði fyrir vakninguna á fallinu sem kallað er á. Ef verið er að kalla á faldað fall mun nýji hlaðinn að hluta til hafa sameiginlegar breytur með gamla hlaðanum sem felst í því að neðri hluti beggja hlaðanna endar á sameiginlegri keðju af breytum. Í þýðandaverkefninu í TÖL202M er ekki endilega nauðsynlegt að nýta þennan möguleika.

4 Gildi í Morpho

Öll gildi í Morpho eru tilvísanir á Java hluti eða eru null. Í keyrandi Morpho forriti er sýndargisti (*virtual register*) ac[1], sem notað er til að skila gildi hverrar segðar. Þegar föll í Morpho skila gildi er gildinu því skilað í (sýndar)gistinu ac.

5 Framhöld

Til að hafa djúpan skilning á virkni Morpho keyrsluumhverfisins er nauðsynlegt að skilja hvernig framhöld (*continuations*) virka. Framhald er gildi sem stendur fyrir þá vinnslu sem eftir er að framkvæma þegar gildi er skilað úr kalli á fall. Í vissum skilningi má líta á framhald sem fall sem tekur eitt viðfang. Viðfangið er þá skilagildið úr fallinu sem verið er að kalla á og keyrsla framhaldsins felst í að framkvæma allt það sem eftir er að framkvæma þegar gildinu er skilað.

Í Morpho er framhald útfært sem gildi sem inniheldur eftirfarandi:

- Skipanafylki (sýndargisti code)
- Staðsetning í skipanafylki (sýndargisti pc)
- Gildahlaði (sýndargisti stack)
- Eðlilegt framhald (sýndargisti ret)
- Afbrigðilegt framhald (sýndargisti ex)

Athugið að framhaldið inniheldur allt sem talið var upp að ofan sem skilgreindi andartaksstöðu í keyrslu þráðlings, nema hvað skilagildið (sýndargistið ac) er ekki með í framhaldinu, enda mun það gildi koma frá fallinu sem framhaldið er framhald fyrir. Það ætti því að vera ljóst að ef við höfum framhald í höndunum og einnig eitthvert gildi þá getum við búið til þráðling úr þeim upplýsingum.

Athugið að bæði framhald og andartaksstaðan í keyrslu þráðlings inniheldur *tvö* framhöld, eðlilega framhaldið (ret) og afbrigðilega framhaldið (ex). Eðlilega framhaldið er það sem tekur við skilagildi fallsins sem verið er að keyra þegar fallið skilar gildi á eðlilegan hátt, en ef upp kemur keyrsluvilla eða ef kastað er afbrigði (*throw* er framkvæmt) þá tekur afbrigðilega framhaldið við niðurstöðunni (sem er þá Java Exception eða eitthvað gildi sem er viðfang í *throw*).

[1] Á ensku er nafnið *accumulator* oft notað yfir slíkt gisti.

6 Vakningarfærslur

Glöggur lesandi mun nú hafa áttað sig á því að vakningarfærlsla í Morpho samanstendur af eftirfarandi:

- Tvö framhöld, eðlilega framhaldið (ret) og afbrigðilega framhaldið (ex). Hvert framhald inniheldur vendivistfang (code og pc) og allar upplýsingar um vakningarfærslu þess falls sem snúa skal til baka til (stýrihlekk).
- Efri hluti gildahlaðans, sem inniheldur milliniðurstöður, staðværar breytur og viðföng núverandi bálks.
- Tengihlekk, sem er tilvísunin í neðsta hluta hlaðans, sem inniheldur breytur í efri földunarhæðum.

Allar þessar upplýsingar eru geymdar í kösinni. Við munum frá TÖL304G að vakningarfærslurnar innihalda eftirfarandi:

- Viðföng
- Vendivistfang
- Stýrihlekk
- Tengihlekk
- Staðværar breytur og milliniðurstöður

Sem einnig má flokka svona:

- Framhald sem samanstendur af stýrihlekki og vendivistfangi.
- Umhverfi sem samanstendur af staðværum breytum, milliniðurstöðum, núverandi viðföngum og tengihlekki.

7 Tilvísanir í breytur

Íhugum Morpho fall g í efstu földunarhæð.

```
g =
    fun(x1,x2 ,..., xN)
    {
        var y1 ,..., yM
        ...
    }
```

Þegar komið er inn í fallið inniheldur gildahlaðinn viðföngin x1,x2,...,xN. Fyrstu skipanirnar sem framkvæmdar eru inni í fallinu bæta breytunum y1,y2,...,yM á hlaðann. Morpho vélarmálsskipanirnar Fetch og Store eru notaðar til að sækja gildi úr breytum og setja gildi í breytur. Hluti af hverri Fetch og Store skipun er staðsetning breytunnar, talið frá botni gildahlaðans. Skipunin (Fetch 0) í fallinu að ofan sækir gildi breytunnar x1 og setur það í sýndargistið ac, skipunin (Store 0) setur nýtt gildi í breytuna x1, sem fengið er úr ac. Skipunin (Fetch N) sækir gildi breytunnar y1, og svo framvegis.

Breyta	**Vísir**
x1	0
...	...
xN	N-1
y1	N
...	...
yM	N+M-1

Tafla 1: Breytutilvísanir í Morpho sýndarvél.

8 Þulur fyrir ýmsar Morphosegðir

Í þýðandaverkefninu hafið þið frjálsar hendur með málfræði málsins sem þið búið til þýðanda fyrir, innan þeirra marka að málið eigi að vera samhæft Morpho, þ.e. að unnt sé að kalla á einingar sem skrifaðar eru í ykkar máli úr Morpho og öfugt. Það þýðir í stórum dráttum að kallrunur (*calling sequence*) í ykkar forritunarmáli þurfa að vera eins og kallrunur í Morpho.

Í eftirfarandi munum við til einföldunar setja okkur þá reglu að allar segðir skili gildi (hvort sem það er notað eða ekki), og að gildinu sé ávallt skilað í sýndargistið ac, eins og föll skila gildi í Morpho.

8.1 Kallsegðir

Morphosegðina

g(s1 ,..., sN)

þar sem s1,...,sN eru Morphosegðir og g er fall í efstu földunarhæð má þýða í eftirfarandi smalamálsrunu:

```
Þ(s1)              ;;; þýðingin á s1
(Push)
 ...
Þ(sN−1)            ;;; þýðingin á sN−1
(Push)
Þ(sN)              ;;; þýðingin á sN
(Call #"g[fN]" N)
```

Í staðinn fyrir Þ[s1] kemur þýðingin á segðinni s1, o.s.frv.

Morpho vélarmálsskipunin (Push) tekur gildið í sýndargistinu ac og ýtir því á hlaðann, sem stækkar um eitt sæti.

Morpho vélarmálsskipunin (Call #"h[fM]" M), þar sem M er heiltala, kallar á fallið h[fM] með M viðföngum. Viðföngin geta verið núll eða fleiri og eru fengin af hlaðanum og úr ac. Gildið í ac verður síðasta viðfangið, ef eitthvert er. Viðföngin eru fjarlægð af núverandi hlaða (ef um fleiri en eitt viðfang er að ræða) og sett á nýjan hlaða áður en byrjað er að framkvæma skipanir inni í nýja fallinu. Nýja vakningin fyrir fallið sem kallað er á mun fá framhald sem heldur áfram keyrslu í næstu skipun á eftir Call skipuninni. Það framhald tekur við þegar vélarmálsskipunin Return er framkvæmd inni í fallinu sem kallað er á.

8.2 Breytusegðir

Morphosegðin

x

þar sem x er staðvær breyta, þýðist í

(**Fetch** n)

þar sem n er staðsetning breytunnar.
Morphosegðin

x = s

þar sem x er breyta og s er segð, þýðist í

Þ[s]
(**Store** n)

8.3 If-segðir

Segðin

if (s1){s2} **else** {s3}

þýðist í

```
        Þ[s1]
        (GoFalse _L1)
        Þ[s2]
        (Go _L2)
_L1:
        Þ[s3]
_L2:
```

Önnur afbrigði if-segða þýðast á sambærilegan hátt.

8.4 While-segðir

Segðin

while(s1){s2}

þýðist í

_L1:
Þ[s1]
(**GoFalse** _L2)
Þ[s2]
(**Go** _L1)
_L2:

8.5 Lesfastasegðir

Segðin

123

þýðist í

(**MakeVal** 123)

Svipað gildir fyrir aðrar gerðir lesfasta svo sem fleytitölur, strengi, staffasta, boolsku lesfastana true og false, ásamt null. Eftirfarandi eru önnur dæmi um þýðingar á lesfastasegðum:

(**MakeVal** 12.3)
(**MakeVal** "abc")
(**MakeVal** 'a')
(**MakeVal** true)
(**MakeVal** false)
(**MakeVal** null)

Vélarmálsskipunin MakeVal setur viðkomandi gildi, sem er hluti af skipuninni, í sýndargistið ac. Þaðan getum við síðan ýtt gildinu á hlaðann, notað það sem viðfang í kalli á fall, sett það í breytu, og svo framvegis.

8.6 Og-segðir

Segðin

s1 && s2

þýðist í

Þ[s1]
(**GoFalse** _L1)
Þ[s2]
_L1:

Spurning: Hvernig þýðist þá eftirfarandi segð?

```
s1 && s2 && s3
```

Skiptir máli hvort og-aðgerðin er tengin til hægri eða vinstri varðandi þulu eða endanlega niðurstöðu?

8.7 Eða-segðir

Segðin

```
s1 || s2
```

þýðist í

```
    Þ[s1]
    (GoTrue _L1)
    Þ[s2]
L1:
```

Spurning: Hvernig þýðist þá eftirfarandi segð?

```
s1 || s2 || s3
```

Skiptir máli hvort eða-aðgerðin er tengin til hægri eða vinstri varðandi þulu eða endanlega niðurstöðu?

8.8 Ekki-segðir

Segðin

```
! s
```

þýðist í

```
Þ[s]
(Not)
```

Spurning: Hvað með eftirfarandi segðir? Fáum við sömu þulu? Sömu merkingu?
Annars vegar:

```
!(s1 && s2)
```

Og hins vegar:

```
(!s1) || (!s2)
```

8.9 Aðrar segðir

Í Morpho eru ýmsar aðrar segðir svo sem return-segð, stefgildissegð til að búa til lokanir, segð til að kalla á lokun, og break-segð til að hætta í lykkju. E.t.v. verður rætt um þær seinna, en hafið í huga að þið eruð ekki bundin af því að nota nákvæmlega sömu segðir og í Morpho. Þið þurfið þó að sjá til þess að ykkar forritunarmál sé nægilega öflugt til þess að unnt sé raunverulega að forrita í því.

8.9.1 go-föll

Við getum tiltölulega auðveldlega (jafnvel í NanoMorpho, sjá blaðsíðu 18) bætt við ýmissi virkni svo sem ræsingu samskeiða falla. Til dæmis gætum við þýtt segðina

```
go h(e1 ,..., eN)
```

í

```
    (Go _L3)
_L1:
    (MakeClosure 0 0 N _L2)
    (Fetch N−1) ;;; aðeins ef N>0
    (Drop 1)    ;;; aðeins ef N>0
    (CallR #"h[fN]" N)
_L2:
    (CallR #" startTask [f1]" 1)
_L3:
    Þ[e1]
    (Push)
     ...
    Þ[eN−1]
    (Push)
    Þ[eN]
    (Call _L1 N)
```

Þetta er svipuð virkni og boðið er upp á í forritunarmálinu **Go**, þar sem þetta er kallað *goroutine*. Eins og í **Go** er reiknað fyrst úr segðunum e1,...,eN í upphaflegum þræði og síðan er ræstur nýr þráður þar sem keyrsla nýja þráðarins felst í að kalla á fallið h með þeim viðföngum sem búið er að reikna.

Annar möguleiki, sem gefur eilítið einfaldari og auðskiljanlegri þulu, er að láta nýja þráðinn reikna segðirnar e1,...,eN. Takið eftir að þetta veldur breytingu á merkingu, sem stundum getur skipt máli, ef hliðarverkanir gerast í einhverjum af segðunum e1,...,eN eða ef forsendur fyrir útkomunum breytast annars staðar áður en búið er að klára að reikna segðirnar. Þá þýðist

```
go h(e1 ,..., eN)
```

í

```
    (MakeClosure 0 0 M L1)
    Þ[e1]
    (Push)
     ...
    Þ[eN−1]
    (Push)
    Þ[eN]
    (CallR #"h[fN]" N)
_L1:
    (Call #" startTask [f1]" 1)
```

Þar sem M er heildarfjöldi breyta í núverandi falli, þ.e. fjöldi viðfanga plús fjöldi staðværra breyta. Hér er merkingin ekki sú sama og í **Go**.

8.9.2 Samskeiða bálkar

Við getum líka notað hugmyndina um go-föll að ofan og einfaldað hana og aðlagað betur að Morpho með eftirfarandi þýðingarmöguleika. Þá þýðum við segðina

```
go { e1; ... eN; }
```

í

```
(MakeClosure 0 0 M L1)
Þ[e1]
 ...
Þ[eN]
(Return)
_L1:
(Call #" startTask [f1]" 1)
```

Þar sem M er heildarfjöldi breyta í núverandi falli, þ.e. fjöldi viðfanga plús fjöldi staðværra breyta.

8.9.3 Viðfangalausar lokanir

Við getum líka aðlagað hugmyndina enn betur að Morpho með eftirfarandi þýðingarmöguleika. Þá þýðum við segðina

```
fun(){ e1; ... eN; }
```

í

```
(MakeClosure 0 0 M L1)
Þ[e1]
 ...
Þ[eN]
(Return)
_L1:
```

Þar sem M er heildarfjöldi breyta í núverandi falli, þ.e. fjöldi viðfanga plús fjöldi staðværra breyta.

Útkoman úr þessari segð er viðfangalaus lokun sem er viðeigandi viðfang í fallið startTask. Sé þessi möguleiki til staðar getum við ræst samskeiða fall svona:

```
startTask (fun(){ e1; ... eN; });
```

Einnig væri hægt að leyfa köll á slíkar lokanir, þannig að ef f væri staðvær breyta mætti leyfa segðina

```
f ();
```

sem væri þá kall á lokun sem yrði þá að vera í breytunni f. Þýðingin á segðinni

```
f ();
```

væri þá

```
(Fetch loc)
(CallClosure 0)
```

þar sem loc er staðsetning breytunnar f.

9 Heildarþýðing stefs

Stefið

```
f =
  fun(x1 ,..., xN)
  {
    var y1=g1 ,..., yM=gM;
    s1;
     ...
    sP;
  };
```

þýðist í

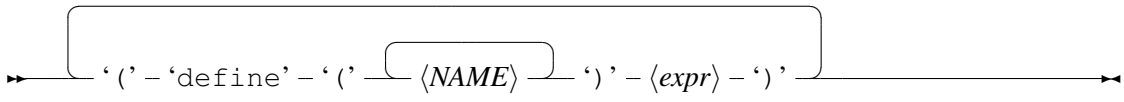
```
#"f[fN]" =
    [
    Þ[g1]
    (Push)
     ...
    Þ[gM]
    (Push)
    Þ[s1]
     ...
    Þ[sP]
    (Return)
    ]
```

10 Dæmi um forritunarmál

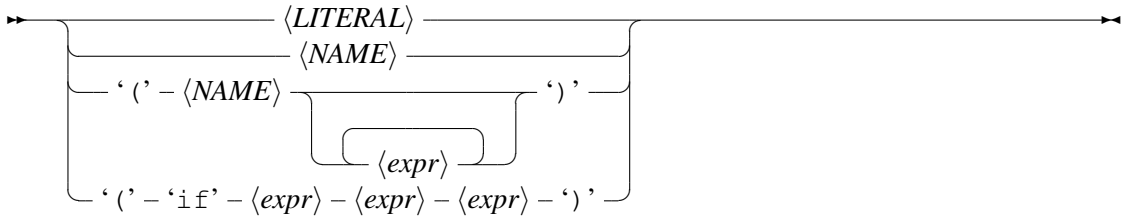
Eftirfarandi málrit sýna dæmi um miseinföld forritunarmál sem gefa vonandi hugmyndir fyrir forritunarmál fyrir þýðandaverkefnið. Forritunarmálið ykkar skal skilgreina í samráði við mig, til þess að það sé hæfilega mikil vinna að skrifa þýðandann.

10.1 Ofureinfalt forritunarmál í Lisp stíl

⟨*program*⟩**:**



⟨*expr*⟩**:**



Hér eru málfræðifyrirbærin ⟨*NAME*⟩ og ⟨*LITERAL*⟩ óskilgreind því reiknað er með því að þau séu skilgreind annars staðar.

10.1.1 Þýðingar

Heildarþýðing forrits. Forrit í þessu forritunarmáli er runa af fallsskilgreiningum.

```
(define (f1 ...) ...)
...
(define (fN ...) ...)
```

þýðist í

```
" forrit .mexe" = main in
!
{{
Þ[(define (f1 ...) ...)]
 ...
Þ[(define (fN ...) ...)]
}}
*
BASIS
;
```

Þýðing fallsskilgreiningar.

```
(define (g x1 ... xN) e)
```

þýðist í

```
#"g[fN]" =
[
    Þ[e]
    (Return)
];
```

Þýðing lesfasta.

123

þýðist í

(**MakeVal** 123)

Svipað gildir fyrir aðra lesfasta, að því tilskildu að Morpho smalamálið kannist við þá.

Þýðing breytutilvísunar.

x

þýðist í

(**Fetch** K)

Þar sem K er staðsetning breytunnar x á gildahlaðanum.

Þýðing kalls.

(g e1 ... eN−1 eN)

þýðist í

```
Þ[e1]
(Push)
 ...
Þ[eN−1]
(Push)
Þ[eN]
(Call #"g[fN]" N)
```

Þýðing if segðar.

(if e1 e2 e3)

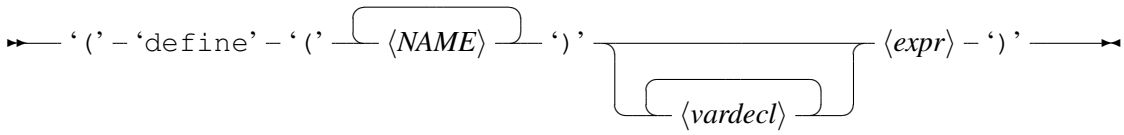
þýðist í

```
        Þ[e1]
        (GoFalse _L1)
        Þ[e2]
        (Go _L2)
_L1:
        Þ[e3]
_L2:
```

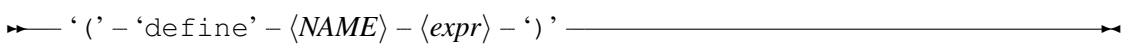
10.2 NanoLisp: Einfalt forritunarmál í Lisp stíl

⟨*program*⟩**:**

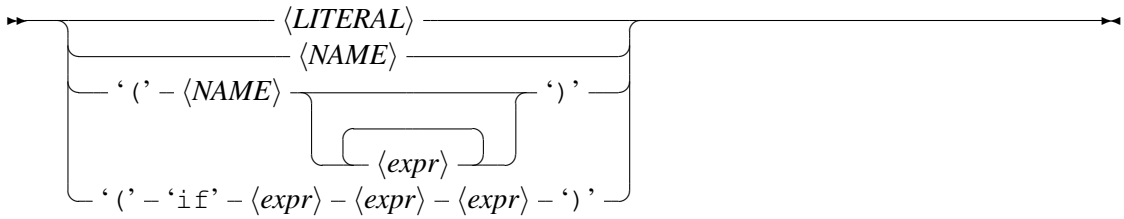
⟨*fundecl*⟩**:**



⟨*vardecl*⟩**:**



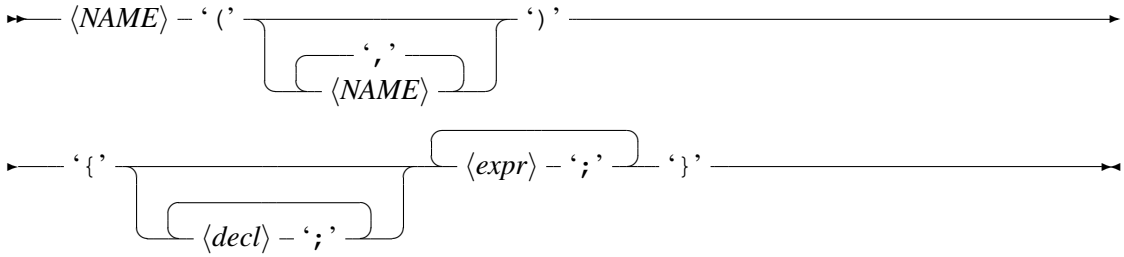
⟨*expr*⟩**:**



10.3 NanoMorpho: Einfalt forritunarmál í Morpho stíl

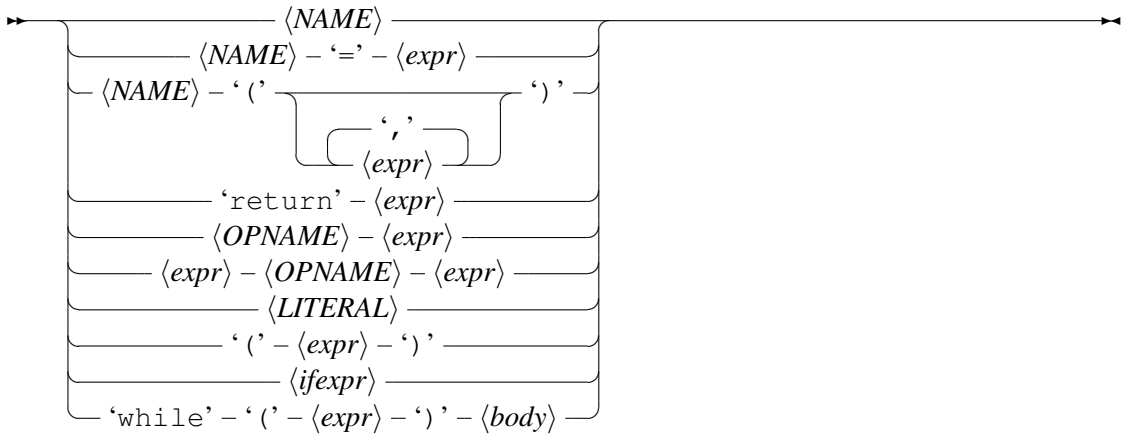
⟨*program*⟩**:**

⟨*function*⟩**:**

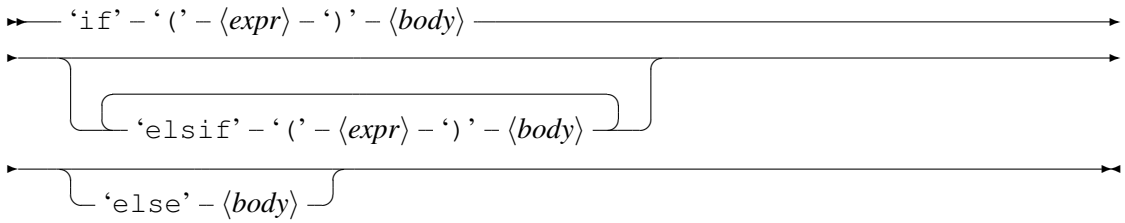


⟨*decl*⟩:

⟨*expr*⟩:



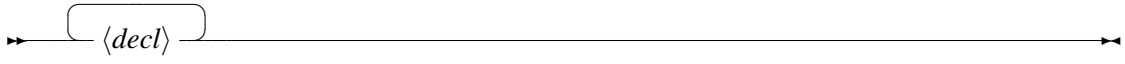
⟨*ifexpr*⟩:



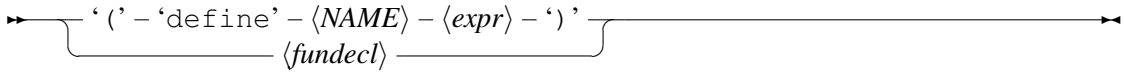
⟨*body*⟩:

10.4 MicroLisp: Flóknara forritunarmál í Lisp stíl

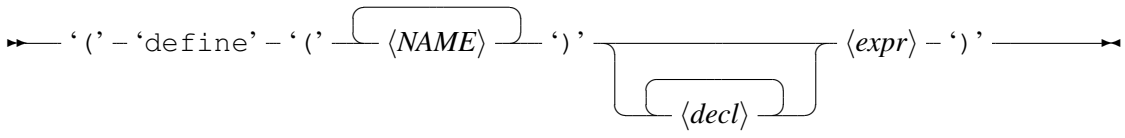
⟨*program*⟩:



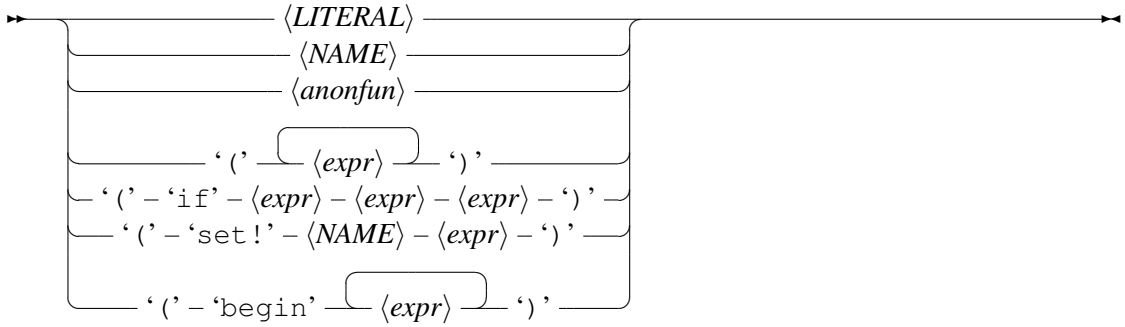
⟨*decl*⟩:



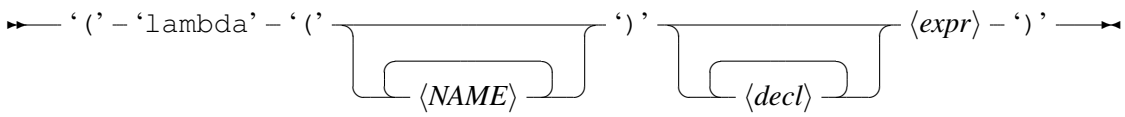
⟨*fundecl*⟩**:**



⟨*expr*⟩**:**



⟨*anonfun*⟩**:**

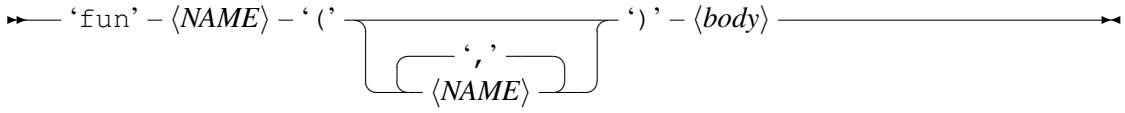


Þetta mál er flóknara í þýðingu því hér höfum við breytur í efstu földunarhæð og möguleika á skilgreiningu og notkun á lokunum ásamt földuðum föllum. Einnig er hægt að gefa breytum ný gildi og keyra bálka með hliðarverkunum, en það, hvort tveggja, er reyndar ekki erfitt í útfærslu.

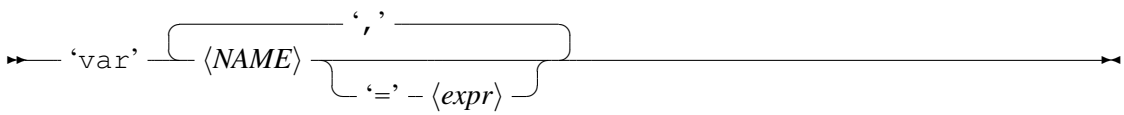
10.5 MicroMorpho: Flóknara forritunarmál í Morpho stíl

⟨*program*⟩**:**

⟨*fundecl*⟩**:**



⟨*vardecl*⟩**:**

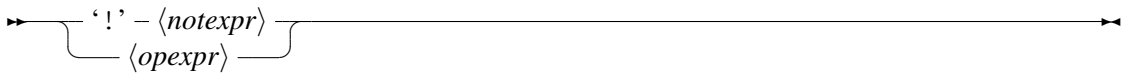


⟨*expr*⟩**:**

⟨*orexpr*⟩**:**

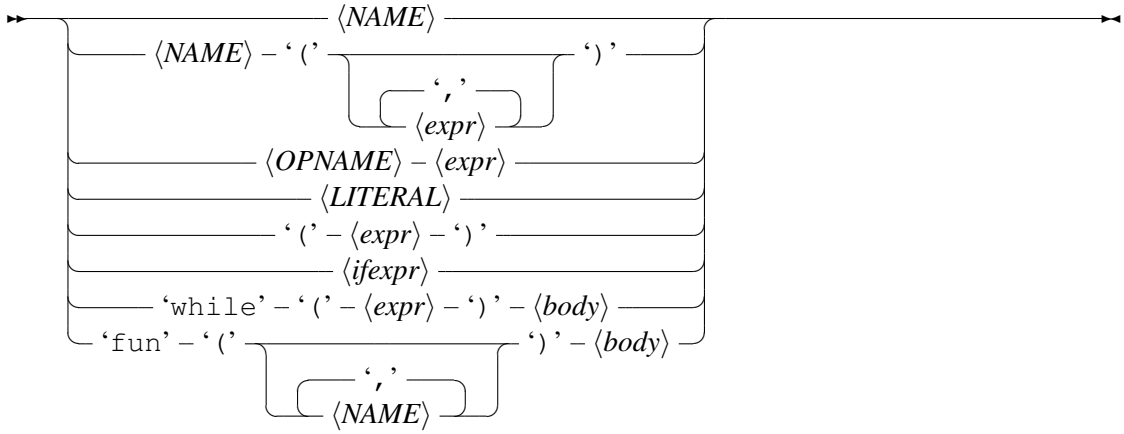
⟨*andexpr*⟩**:**

⟨*notexpr*⟩**:**

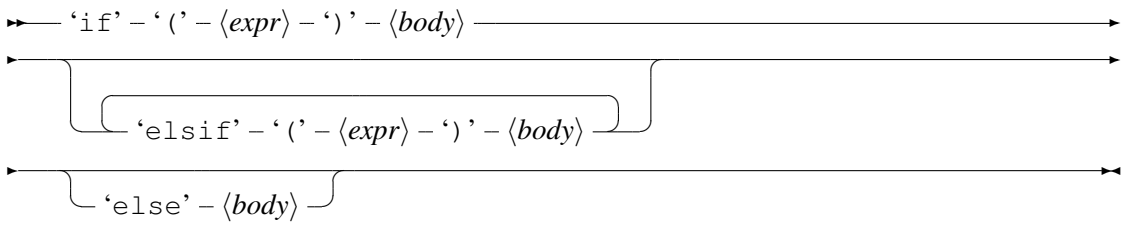


⟨*opexpr*⟩**:**

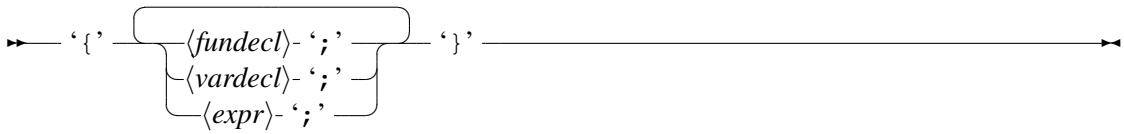
⟨*smallexpr*⟩**:**



⟨*ifexpr*⟩:



⟨*body*⟩:



11 Helstu Morpho vélarmálsskipanir

Eftirfarandi eru lýsingar á þeim Morpho vélarmálsskipunum sem líklegast er að séu nytsamlegar í þýðandaverkefninu. Sumir þýðendur munu aðeins nota hluta af þessum skipunum. Aðrir munu e.t.v. nota fleiri skipanir, einkum ef forritunarmálið býður upp á földuð föll og föll sem gildi.

11.1 Call

Notkun: (Call #"g[fN]" N M) eða (Call #"g[fN]" N)

Fyrir: N og M eru heiltölur, N,M≥ 0. Gildahlaðinn inniheldur a.m.k. M+N-1 gildi. #"g[fN]" vísar á fall sem tekur N viðföng og notar M breytur í efri földunarhæðum. Morpho smalinn breytir reyndar skipanaviðfangi á þessu sniði í heiltölu. Sú heiltala er afstætt (emphrelative) vistfang fyrstu skipunar í fallinu sem verið er að kalla á, þ.e. fjarlægðin (jákvæð eða neikvæð) milli Call skipunarinnar og fyrstu skipunar fallsins. Skipunin (Call #"g[fN]" N) er jafngild skipuninni (Call #"g[fN]" N 0).

Eftir: Áhrif þessarar skipunar er að færa keyrsluna yfir í fallið sem #"g[fN]" vísar á. En áður en það er gert eru viðföngin fyrir fallið fjarlægð af gildahlaðanum í núverandi falli og sett á nýjan gildahlaða fyrir nýja fallið. Neðsti hluti þessa nýja gildahlaða hefur M sameiginlegar breytur með gildahlaðanum í vakningunni sem kallar. Einnig er smíðað nýtt framhald (ret) sem inniheldur allar nauðsynlegar upplýsingar til að snúa til baka í núverandi fall. Þetta framhald verður eðlilega framhaldið í keyrslu nýja fallsins. Heildaráhrifin valda því breytingu á gildahlaðanum (stack) ásamt sýndargistunum pc og ret.

Dæmi: Til að reikna Morpho segðina 1+2 notum við skipanirnar

(**MakeVal** 1)
(**Push**)
(**MakeVal** 2)
(**Call** #"+[f2]" 2)

eða (jafngilt)

(**MakeVal** 1)
(**MakeValP** 2)
(**Call** #"+[f2]" 2)

11.2 CallR

Notkun: (CallR #"g[fN]" N M) eða (CallR #"g[fN]" N)

Fyrir: Sama og fyrir skipunina (Call #"g[fN]"N M) eða (Call #"g[fN]"N), eftir atvikum.

Eftir: Eins og fyrir (Call #"g[fN]" N M) nema hvað ekki er búið til nýtt framhald, ret, heldur er sama framhald notað. Áhrif þessa eru að kallið á nýja fallið er halaendurkvæmt kall sem mun ekki snúa til baka í núverandi kall heldur til þess kalls sem átti að fá niðurstöðuna úr núverandi kalli. Áhrifin af skipuninni eru svipuð og áhrifin af skipunum

(**Call** #"g[fN]" N M)
(**Return**)

nema hvað að engin lenging verður á stýrihlekkjakeðjunni (keðjunni af framhöldum) við framkvæmdina á CallR og einnig er CallR skipunin einfaldari í framkvæmd og hraðvirkari ein og sér heldur en skipunin Call ein og sér.

Dæmi: Til að reikna Morpho segðina return 1+2 notum við skipanirnar

(**MakeVal** 1)
(**Push**)
(**MakeVal** 2)
(**Call** #"+[f2]" 2)
(**Return**)

eða (jafngilt)

(**MakeVal** 1)
(**Push**)
(**MakeVal** 2)
(**CallR** #"+[f2]" 2)

eða (jafngilt)

(**MakeVal** 1)
(**MakeValP** 2)
(**CallR** #"+[f2]" 2)

11.3 CallClosure

Notkun: (CallClosure N)

Fyrir: N er heiltala, N≥ 0. Gildahlaðinn inniheldur a.m.k. N gildi. Gildin sem eru á hlaðanum og síðan í ac, eru, í röð, fyrst lokun sem stendur fyrir fall með N viðföng, og síðan viðföngin, N talsins. Athugið að sértilfelli er fall sem tekur núll viðföng, og þá er lokunin í ac.

Eftir: Áhrif þessarar skipunar er að færa keyrsluna yfir í fallið sem lokunin vísar á. En áður en það er gert eru viðföngin fyrir fallið fjarlægð af gildahlaðanum í núverandi falli og einnig lokunin, ef hún er á hlaðanum (þ.e. ef N>0) og sett á nýjan gildahlaða fyrir nýja fallið. Einnig er smíðað nýtt framhald (ret) sem inniheldur allar nauðsynlegar upplýsingar til að snúa til baka í núverandi fall. Þetta framhald verður eðlilega framhaldið í keyrslu nýja fallsins. Heildaráhrifin valda því breytingu á gildahlaðanum (stack) ásamt sýndargistunum code, pc og ret.

11.4 CallClosureR

Notkun: (CallClosureR N)

Fyrir: Eins og fyrir (CallClosure N).

Eftir: Eins og fyrir (CallClosure N) nema hvað ekki er búið til nýtt framhald, ret, heldur er sama framhald notað. Áhrif þessa eru að kallið á nýja fallið er halaendurkvæmt kall sem mun ekki snúa til baka í núverandi kall heldur til þess kalls sem átti að fá niðurstöðuna úr núverandi kalli. Áhrifin af skipuninni eru svipuð og áhrifin af skipunum

(**CallClosure** N)
(**Return**)

nema hvað að engin lenging verður á stýrihlekkjakeðjunni (keðjunni af framhöldum) við framkvæmdina á CallClosureR og einnig er CallClosureR skipunin einfaldari í framkvæmd og hraðvirkari ein og sér heldur en skipunin CallClosure ein og sér.

11.5 Drop

Notkun: (Drop K)

Fyrir: Gildahlaðinn inniheldur a.m.k. K gildi.

Eftir: K gildi hafa verið fjarlægð af hlaðanum.

11.6 Fetch

Notkun: (Fetch K)

Fyrir: K er heiltölufasti, K$\geq$0, sem vísar á staðsetningu á gildahlaðanum. Ef N er fjöldi gilda á gildahlaðanum verður að gilda 0$\leq$K$<$N. Staðsetning K=0 stendur fyrir neðsta sæti, staðsetning N-1 fyrir efsta sæti á hlaðanum.

Eftir: Sýndargistið ac hefur fengið nýtt gildi, sem er gildi sem sótt var úr sæti K á gildahlaðanum.

11.7 FetchP

Notkun: (Fetch K)

Fyrir: Sama forskilyrði og fyrir (Fetch K).

Eftir: Áhrifin af (FetchP K) eru þau sömu og fyrir skipanirnar

(**Push**)
(**Fetch** K)

11.8 FetchR

Notkun: (FetchR K)

Fyrir: Sama forskilyrði og fyrir (Fetch K).

Eftir: Áhrifin af (FetchR K) eru þau sömu og fyrir skipanirnar

(**Fetch** K)
(**Return**)

11.9 Go

Notkun: (Go lab)

Fyrir: lab er heiltala sem er afstætt vistfang á vélarmálsskipun í núverandi fylki af vélarmálsskipunum.

Eftir: Ef Go skipunin er í sæti K í fylkinu code, þá eru áhrifin af (Go P) þau að gefa gistinu pc gildið K+lab+1, sem verður Þá vísirinn á næstu vélarmálsskipun sem framkvæmd verður. Athugið að í flestum Morpho vélarmálsskipunum er sýndargistið pc stækkað um einn. Undantekningar eru þær skipanir sem ætlaðar eru til að stýra atburðarásinni, s.s. Go, Call, Return og fleiri skyldar skipanir. Morpho smalinn gefur kost á að nota merki (*label*) í stað heiltölufasta fyrir lab, en á endanum er sett heiltala í staðinn, sem reiknuð er á þann hátt að merkingin verði sú sem okkur hentar.

11.10 GoFalse

Notkun: (GoFalse lab)

Fyrir: Sama og fyrir (Go lab).

Eftir: Sama og fyrir (Go lab), nema hvað aðeins er stokkið ef gildið í sýndargistinu ac er ósatt. Ósönn gildi eru tóma tilvísunin null og boolska gildið false. Öll önnur gildi eru sönn. Takið eftir að heiltalan 0 er sönn, ekki ósönn.

11.11 GoTrue

Notkun: (GoTrue lab)

Fyrir: Sama og fyrir (Go lab).

Eftir: Sama og fyrir (Go lab), nema hvað aðeins er stokkið ef gildið í sýndargistinu ac er satt. Ósönn gildi eru tóma tilvísunin null og boolska gildið false. Öll önnur gildi eru sönn.

11.12 MakeClosure

Notkun: (MakeClosure funlab N K pastlab)

Fyrir: funlab er afstætt vistfang (yfirleitt er notað merki (*label*)) sem vísar á fyrstu skipun í falli. N er fjöldi viðfanga sem fallið tekur. K er fjöldi þeirra breyta í efri földunarhæðum sem þetta fall og núverandi fall hafa sameiginlegar. pastlab er afstætt vistfang (merki, *label*) Morpho vélarmálsskipunar.

Eftir: Búið er að smíða lokun sem vísar á hið umrædda fall. Lokunin inniheldur tengihlekk sem vísar í breyturnar K sem fallið og núverandi fall hafa til samans. Eftir þessa smíð er haldið áfram keyrslu í skipuninni sem pastlab vísar til.

11.13 MakeClosureP

Notkun: (MakeClosureP funlab N K pastlab)

Fyrir: Eins og fyrir (MakeClosure funlab N K pastlab).

Eftir: Áhrifin af (MakeClosureP funlab N K pastlab) eru þau sömu og fyrir skipanirnar

(**Push**)
(**MakeClosure** funlab N K pastlab)

11.14 MakeClosureR

Notkun: (MakeClosureR funlab N K pastlab)

Fyrir: Eins og fyrir (MakeClosure funlab N K 0).

Eftir: Áhrifin af (MakeClosureR funlab N K pastlab) eru þau sömu og fyrir skipanirnar

(**MakeClosure** funlab N K 0)
(**Return**)

sem er sama og

(**MakeClosure** funlab N K pastlab)
pastlab :
(**Return**)

Athugið að gildið á pastlab skiptir ekki máli heldur er alltaf framkvæmt jafngildi (Return) skipunarinnar.

11.15 MakeVal

Notkun: (MakeVal L)

Fyrir: L er heiltölufasti, fleytitölufasti, staffasti, strengfasti, sanngildi (true eða false) eða null.

Eftir: Búið er að setja gildið á fastanum L í sýndargistið ac.

Athugasemd: Gildi í Morpho eru ávallt tilvísanir. Heiltölufastar eru til dæmis tilvísanir á hlut af tagi Integer, Long eða BigInteger. Fleytitölufastar eru tilvísanir á hlut af tagi Double. Boolsk fastagildi eru tilvísanir á hlut af tagi Boolean. Staffastar eru tilvísanir á hlut af tagi Char. Tóma tilvísunin null er einnig lögleg. Í skipuninni MakeVal er aðeins hægt að nota ofangreindar gerðir viðfanga.

11.16 MakeValP

Notkun: (MakeValP L)

Fyrir: Sama og fyrir (MakeVal L)

Eftir: Áhrifin af (MakeValP L) eru þau sömu og fyrir skipanirnar

(**Push**)
(**MakeVal** L)

11.17 MakeValR

Notkun: (MakeValR L)

Fyrir: Sama og fyrir (MakeVal L)

Eftir: Áhrifin af (MakeValR L) eru þau sömu og fyrir skipanirnar

(**MakeVal** L)
(**Return**)

11.18 Not

Notkun: (Not)

Fyrir: Ekkert.

Eftir: Sýndargistið ac hefur fengið nýtt boolskt gildi sem er true ef ac var áður ósatt, þ.e. null eða false, en false annars.

11.19 NotP

Notkun: (NotP)

Fyrir: Ekkert.

Eftir: Áhrifin af (NotP) eru þau sömu og fyrir skipanirnar

(**Push**)
(**Not**)

11.20 NotR

Notkun: (NotR)

Fyrir: Ekkert.

Eftir: Áhrifin af (NotR) eru þau sömu og fyrir skipanirnar

(**Not**)
(**Return**)

11.21 Push

Notkun: (Push)

Fyrir: Ekkert.

Eftir: Búið er að ýta gildinu í ac á gildahlaðann.

11.22 Return

Notkun: (Return)

Fyrir: Eðlilega framhaldið, ret, vísar í löglegt framhald. Þetta forskilyrði mun ávallt vera satt í rétt þýddu forriti.

Eftir: Búið er að snúa til baka í það framhald sem ret vísar á. Við höfum þá fengið ný gildi í öll (sýndar)gisti Morpho sýndarvélarinnar nema ac, sem heldur sínu gildi. Við höfum einnig fengið nýjan gildahlaða, eða réttara sagt höfum við fengið aftur gamla gildahlaðann sem er gildahlaðinn sem var við lýði í fallinu sem fær skilagildið (ac) úr núverandi kalli.

11.23 Store

Notkun: (Store K)

Fyrir: K er heiltölufasti, $K \geq 0$, sem vísar á staðsetningu á gildahlaðanum. Ef N er fjöldi gilda á gildahlaðanum verður að gilda $0 \leq K < N$. Staðsetning K=0 stendur fyrir neðsta sæti, staðsetning N-1 fyrir efsta sæti á hlaðanum.

Eftir: Gildið í sýndargistinu ac hefur verið afritað í sæti K á gildahlaðanum.

11.24 StoreP

Notkun: (StoreP K)

Fyrir: Eins og fyrir (Store K).

Eftir: Áhrifin af (StoreP K) eru þau sömu og fyrir skipanirnar

(**Push**)
(**Store** K)

11.25 StoreR

Notkun: (StoreR K)

Fyrir: Eins og fyrir (Store K).

Eftir: Áhrifin af (StoreR K) eru þau sömu og fyrir skipanirnar

(**Store** K)
(**Return**)

12 Dæmi um þýðendur

Eftirfarandi eru tveir þýðendur fyrir ofureinfalt NanoLisp sem lýst er í kafla 10.1 á blaðsíðu 15.

12.1 Þýðandi með endurkvæmri ofanferð

Eftirfarandi forritstexti, í skránni NanoLisp.java, er Java klasi sem er þýðandi fyrir NanoLisp.

Þýðandinn er samsettur af þremur fösum, lesgreini (*scanner, lexical analyser, lexer*), þáttara (*parser, syntax analyser*) sem einnig er milliþulusmiður (*intermediate code generator*), og lokaþulusmið (*final code generator*). Það er mjög algengt að þýðendur séu einmitt saman settir á þennan hátt.

Lesgreinirinn samanstendur af innri klösunum Lexer og LexCase, sem notaðir eru til að bera kennsl á frumeiningarnar í forritunarmálinu, sviga, lesfasta, nöfn og lykilorð, sem öll eru send áfram til þáttarans, ásamt athugasemdum sem þáttarinn fær aldrei að sjá.

Þáttarinn samanstendur af tilviksboðunum í ysta klasanum NanoLisp. Hvert af boðunum program, fundecl og expr ber kennsl á eitt málfræðifyrirbæri og skilar milliþulu fyrir það. Milliþulan samanstendur af Java Object[] fylkjum, sem innihalda allar nauðsynlegar upplýsingar um viðkomandi fyrirbæri.

Þýðandi þessi notar þáttunartækni sem kölluð er **endurkvæm ofanferð** (*recursive descent*) sem er ein þeirra aðferða sem við munum kynnast í TÖL202M. Endurkvæm ofanferð hefur þann kost að ekki þarf nein sérstök tól önnur en almennt forritunarmál til að smíða þáttarann. Forritunarmálið þarf að styðja gagnkvæmt endurkvæm föll, en það gildir um flest forritunarmál nú til dags.

```
// Þýðandi fyrir ofureinfalt NanoLisp forritunarmál.                    1
// Höfundur: Snorri Agnarsson, 2014–2016.                               2
                                                                        3
import java.io.Reader;                                                  4
import java.io.FileReader;                                              5
import java.io.InputStreamReader;                                       6
import java.io.IOException;                                             7
import java.util.regex.Pattern;                                         8
import java.util.regex.Matcher;                                         9
import java.util.Vector;                                                10
                                                                        11
public class NanoLisp                                                   12
{                                                                       13
    // Eftirfarandi fastar standa fyrir allar þær                       14
    // mögulegu gerðir af segðum sem milliþula                          15
    // (intermediate code) getur innihaldið.                            16
    // Þessar fjórar gerðir segða (ásamt þeim                           17
    // möguleika að skrifa föll sem nota slíkar                         18
    // segðir) duga reyndar til að hægt sé að                           19
    // reikna hvað sem er reiknanlegt.                                  20
                                                                        21
    // Tilvik af klasanum LexCase er skilgreining                       22
    // á einhverju lesgreinanlegu fyrirbæri, sem                        23
    // má þá koma fyrir í löglegum forritstexta.                        24
    // Lesgreinirinn í þessum þýðanda er skrifaður                      25
    // á mjög frumstæðan hátt og aðferðin er ekki                       26
    // til eftirbreytni í neinum þýðanda sem ætlaður                    27
    // er til raunverulegrar notkunar.                                  28
    static class LexCase                                                29
    {                                                                   30
        final Pattern pat;                                              31
        final char token;                                               32
        int end;                                                        33
                                                                        34
        // Notkun: LexCase c = new LexCase(p,t);                        35
        // Fyrir:  p er strengur sem inniheldur reglulega               36
        //         segð sem skilgreinir mál sem inniheldur              37
```

```
//          þá strengi sem sem eiga að flokkast sem
//          eitthvert tiltekið lesgreinanlegt
//          fyrirbæri (les, lexeme). t er stafur
//          sem er það tók (token) sem stendur
//          fyrir þetta mengi strengja (þetta mál).
// Eftir:   c vísar á hlut sem nota má til að
//          bera kennsl á strengi í málinu.
public LexCase( String p, char t )
{
    pat = Pattern.compile(p,Pattern.MULTILINE);
    token = t;
}

// Notkun: boolean b = c.match(s,pos);
// Fyrir:  s er strengur og pos er staðsetning
//         innan s.
// Eftir:  b er satt þþaa staðsetningin pos vísi
//         á byrjun hlutstrengs sem er í málinu
//         sem c skilgreinir.
public boolean match( String s, int pos )
{
    Matcher m = pat.matcher(s).region(pos,s.length());
    boolean res = m.lookingAt();
    if( res ) end = m.end();
    return res;
}

// Notkun: int i = c.end();
// Fyrir:  Búið er að kalla c.match(s,pos) og fá
//         sanna niðurstöðu.
// Eftir:  i inniheldur staðsetningu innan s sem
//         vísar á næsta staf á eftir þeim streng
//         sem borið var kennsl á í match kallinu.
public int end()
{
    return end;
}
```

```
}

// Hlutur af tagi Lexer er lesgreinirinn í þessum þýðanda.
// Þetta er ákaflega frumstæður lesgreinir sem getur
// lesgreint allt að 100000 stafa inntaksskrár og notar
// frekar lélega aðferð til þess.
static class Lexer
{
    final LexCase[]
        cases =
            { new LexCase("\\G\\(",'(')
            , new LexCase("\\G\\)",')')
            , new LexCase("\\G(\\s|(;.*$)|\\r|\\n)",'C')
            , new LexCase("\\G\\d+(\\.(\\d)+([eE][+\\-]?\\d+)?)?",'L')
            , new LexCase("\\G\\\"([^\\\\\"]|\\t|\\n|\\r|\\f)*\\\"",'L')
            , new LexCase("\\G\\'([^\\\\']|\\t|\\n|\\r|\\f)\\'",'L')
            , new LexCase("\\G(\\p{Alpha}|[+\\-\\.*/<=>!\\?:$%_&~^0-9])+",'N')
            , new LexCase(".",'?')
            };
    final String input;
    int i;
    char token;
    String lexeme;
    // Fastayrðing gagna:
    //   input inniheldur forritstextann sem verið er að þýða.
    //   i er staðsetning næsta ólesna stafs í forritstextanum.
    //   token er stafur sem stendur fyrir næsta tók sem ekki
    //   er búið að vinna úr.  Það samsvarar stafarunu rétt fyrir
    //   framan staðsetninguna i.  lexeme er strengur sem
    //   inniheldur þá stafi úr input sem samsvara token.
    //   cases inniheldur skilgreiningar á þeim málum sem
    //   lesgreinirinn gerir greinarmun á.  Hvert stak í
    //   cases skilgreinir annars vegar eitthvert mál (mengi
    //   strengja) sem borið er kennsl á og hins vegar staf
    //   sem er samsvarandi tók.  Merking tókanna er:
    //     '(':  Strengurinn "("
    //     ')':  Strengurinn ")"
```

```
//      'C':  Strengur sem er athugasemd eða bilstafur,
//            sem skal því ekki skila áfram til þáttarans
//      'L':  Strengur sem stendur fyrir lesfasta, þ.e.
//            tölufasti, strengfasti, staffasti eða einn
//            af lesföstunum true, false eða null.
//      'N':  Strengur sem er löglegt breytunafn eða nafn
//            á falli.
//      'I':  Lykilorðið if.
//      'D':  Lykilorðið define.
//      '$':  Skrárlok, þ.e. endir inntaksins.
//    Athugið að nokkur breytunöfn eru þess eðlis að þegar
//    lesgreinirinn skilar tóki fyrir þau þá á hann að
//    segja að þau séu lesfastar.  Þetta eru breytunöfnin
//    true, false og null.  Lesgreinirinn þarf því að
//    athuga hvort um þessi sérstöku breytunöfn er að ræða
//    eftir að í ljós hefur komið að lesgreindi strengurinn
//    er breytunafn.  Sama gildir um lykilorðin if og define.

// Notkun: Lexer l = new Lexer(r);
// Fyrir:  r er Reader sem inniheldur allt að 100000
//         stafi.
// Eftir:  l vísar á nýjan lesgreini sem lesgreinir
//         innihaldið í r.  Lesgreinirinn er í upphafi
//         staðsettur á fremsta lesi (lexeme) í r.
public Lexer( Reader r ) throws IOException
{
    StringBuffer b = new StringBuffer();
    char[] buf = new char[100000];
    for(;;)
    {
        int n = r.read(buf);
        if( n == -1 ) break;
        b.append(buf,0,n);
    }
    input = b.toString();
    i = 0;
    advance();
```

```
}

// Notkun: char c = l.getToken();
// Eftir:  c er tókið (token) sem stendur fyrir
//         það mál sem næsta les í l flokkast í.
char getToken()
{
    return token;
}

// Notkun: String s = l.getLexeme();
// Eftir:  s er næsta les í l.
String getLexeme()
{
    return lexeme;
}

// Notkun: l.advance();
// Eftir:  l hefur færst áfram á næsta les í
//         inntakinu (sem ekki er athugasemd).
void advance()
{
    for(;;)
    {
        if( i>=input.length() )
        {
            token = '$';
            lexeme = "EOF";
            return;
        }
        for( int k=0 ; k!=cases.length ; k++ )
        {
            Matcher m = cases[k].pat.matcher(input);
            if( m.find(i) )
            {
                int j = m.end();
                token = cases[k].token;
```

```
                if( token=='C' )
                {
                    i = j;
                    break;
                }
                lexeme = input.substring(i,j);
                i = j;
                if( token=='N' )
                {
                    if( lexeme.equals("if") )
                        token = 'I';
                    else if( lexeme.equals("define") )
                        token = 'D';
                    else if( lexeme.equals("null") )
                        token = 'L';
                    else if( lexeme.equals("true") )
                        token = 'L';
                    else if( lexeme.equals("false") )
                        token = 'L';
                }
                return;
            }
        }
    }
}

// Notkun: String n = Lexer.tokenName(c);
// Fyrir:  c er stafur sem er einn þeirra sem
//         lexgreinirinn getur skilað sem tók.
// Eftir:  n er nafnið á þessu tóki á mannlega
//         læsilegu sniði.
static String tokenName( char t )
{
    switch( t )
    {
    case '(': return "(";
    case ')': return ")";
```

```
        case 'N': return "name";
        case 'L': return "literal";
        case 'D': return "define";
        case 'I': return "if";
        case '$': return "EOF";
        }
        return "?";
    }

    // Notkun: l.over(c);
    // Fyrir:  c er stafur sem stendur fyrir mögulegt tók.
    //         Næsta tók í l er c.
    // Eftir:  Búið er að færa lesgreininn eitt skref áfram
    //         eins og í advance().
    // Afbrigði:  Ef svo vill til að næsta tók í l er ekki
    //            c þá eru skrifuð villuboð og keyrslan stöðvast.
    String over( char tok )
    {
        if( token!=tok ) throw new Error("Expected "+tokenName(tok)+", found "+lexeme);
        String res = lexeme;
        advance();
        return res;
    }
}

// lex er lesgreinirinn.
Lexer lex;
// Inni í hverri fallsskilgreiningu inniheldur vars nöfnin
// á viðföngunum í fallið (þ.e. leppunum eða breytunöfnunum
// sem standa fyrir viðföngin), í sætum 1 og aftar.  Sæti
// 0 inniheldur nafn fallsins sem verið er að skilgreina.
String[] vars;

// Notkun: NanoLisp n = new NanoLisp(l);
// Fyrir:  l er lesgreinir.
// Eftir:  n vísar á nýjan NanoLisp þýðanda sem þýðir inntakið
//         sem l hefur.
```

```
public NanoLisp( Lexer lexer )
{
    lex = lexer;
}

// Notkun: int i = n.varPos(name);
// Fyrir:  n er NanoLisp þýðandi og er að þýða stofn einhvers
//         falls. name er nafnið á einhverju viðfangi í fallið.
// Eftir:  i er staðsetning viðfangsins í viðfangarunu fallsins
//         þar sem fyrsta viðfang er talið vera í sæti 0.
int varPos( String name )
{
    for( int i=1 ; i!=vars.length ; i++ )
        if( vars[i].equals(name) ) return i-1;
    throw new Error("Variable "+name+" is not defined");
}

// Notkun: Object[] code = n.program();
// Fyrir:  n er NanoLisp þýðandi og inntakið er löglegt
//         NanoLisp forrit.
// Eftir:  Búið er að þýða forritið og code vísar á nýtt
//         fylki sem inniheldur milliþulurnar fyrir öll
//         föllin í forritinu.
// Afbrigði: Ef forritið er ekki löglegt þá eru skrifuð
//           villuboð og keyrslan stöðvuð.
Object[] program()
{
    Vector<Object> res = new Vector<Object>();
    while( lex.getToken() == '(' ) res.add(fundecl());
    return res.toArray();
}

// Notkun: Object[] fun = n.fundecl();
// Fyrir:  n er NanoLisp þýðandi sem er staðsettur í
//         byrjun fallsskilgreiningar.
// Eftir:  Búið er að lesa fallsskilgreininguna og fun vísar
//         á nýtt fylki sem er milliþulan fyrir fallið.
```

```
//          Þýðandinn er nú að horfa á næsta tákn í inntakinu
//          fyrir aftan fallsskilgreininguna.
Object[] fundecl()
{
    lex.over('(');
    lex.over('D');
    lex.over('(');
    Vector<String> args = new Vector<String>();
    args.add(lex.over('N'));
    while( lex.getToken()!=')' ) args.add(lex.over('N'));
    lex.over(')');
    vars = new String[args.size()];
    args.toArray(vars);
    Object[] res = new Object[]{vars[0],vars.length-1,expr()};
    vars = null;
    lex.over(')');
    return res;
}

// Notkun: Object[] e = n.expr();
// Fyrir:  n er NanoLisp þýðandi sem er staðsettur í
//         byrjun segðar innan fallsskilgreiningar.
// Eftir:  Búið er að lesa segðina og þýða hana.
//         e vísar á milliþuluna fyrir segðina.
//         Þýðandinn er nú að horfa á næsta tákn í inntakinu
//         fyrir aftan segðina.
Object[] expr()
{
    Object[] res;
    switch( lex.getToken() )
    {
    case 'L':
        res = new Object[]{"LITERAL",lex.getLexeme()};
        lex.advance();
        return res;
    case 'N':
        res = new Object[]{"FETCH",varPos(lex.getLexeme())};
```

```
            lex.advance();                                                                                  334
            return res;                                                                                     335
        case '(':                                                                                           336
            lex.advance();                                                                                  337
            switch( lex.getToken() )                                                                        338
            {                                                                                               339
            case 'N':                                                                                       340
                String name = lex.over('N');                                                                341
                Vector<Object> args = new Vector<Object>();                                                 342
                while( lex.getToken()!=')' ) args.add(expr());                                              343
                lex.advance();                                                                              344
                return new Object[]{"CALL",name,args.toArray()};                                            345
            case 'I':                                                                                       346
                Object cond,thenexpr,elseexpr;                                                              347
                lex.advance();                                                                              348
                cond = expr();                                                                              349
                thenexpr = expr();                                                                          350
                elseexpr = expr();                                                                          351
                lex.over(')');                                                                              352
                return new Object[]{"IF",cond,thenexpr,elseexpr};                                           353
            }                                                                                               354
            throw new Error("Expected␣a␣name␣or␣the␣keyword␣'if',␣found␣"+lex.getLexeme());                 355
        default:                                                                                            356
            throw new Error("Expected␣an␣expression,␣found␣"+lex.getLexeme());                              357
        }                                                                                                   358
}                                                                                                           359
                                                                                                            360
// Notkun: emit(line);                                                                                      361
// Fyrir:  line er lína í lokaþulu.                                                                         362
// Eftir:  Búið er að skrifa línuna á aðalúttak.                                                            363
static void emit( String line )                                                                             364
{                                                                                                           365
    System.out.println(line);                                                                               366
}                                                                                                           367
                                                                                                            368
// Notkun: generateProgram(name,p);                                                                         369
// Fyrir:  name er strengur, p er fylki fallsskilgreininga,                                                 370
```

```
//          þ.e. fylki af milliþulum fyrir föll.
// Eftir:   Búið er að skrifa lokaþulu fyrir forrit sem
//          samanstendur af föllunum þ.a. name er nafn
//          forritsins.
static void generateProgram( String name, Object[] p )
{
    emit("\""+name+".mexe\" = main in");
    emit("!{{");
    for( int i=0 ; i!=p.length ; i++ ) generateFunction((Object[])p[i]);
    emit("}}*BASIS;");
}

// Notkun: generateFunction(f);
// Fyrir:  f er milliþula fyrir fall.
// Eftir:  Búið er að skrifa lokaþulu fyrir fallið á
//         aðalúttak.
static void generateFunction( Object[] f )
{
    // f = {fname, argcount, expr}
    String fname = (String)f[0];
    int count = (Integer)f[1];
    emit("#\""+fname+"[f"+count+"]\" =");
    emit("[");
    generateExprR((Object[])f[2]);
    emit("];");
}

static int nextLab = 1;

// Notkun: int i = newLab();
// Eftir:  i er jákvæð heiltala sem ekki hefur áður
//         verið skilað úr þessu falli.  Tilgangurinn
//         er að búa til nýtt merki (label), sem er
//         ekki það sama og neitt annað merki.
static int newLab()
{
    return nextLab++;
```

```
}

// Notkun: generateExpr(e);
// Fyrir:  e er milliþula fyrir segð.
// Eftir:  Búið er að skrifa lokaþulu fyrir segðina
//         á aðalúttak.  Lokaþulan reiknar gildi
//         segðarinnar og skilur gildið eftir í
//         gildinu ac.
static void generateExpr( Object[] e )
{
    switch( (String)e[0] )
    {
    case "FETCH":
        // e = {"FETCH",pos}
        emit("(Fetch␣"+e[1]+")");
        return;
    case "LITERAL":
        // e = {"LITERAL",literal}
        emit("(MakeVal␣"+(String)e[1]+")");
        return;
    case "IF":
        // e = {"IF",cond,then,else}
        int labElse = newLab();
        int labEnd = newLab();
        generateJump((Object[])e[1],0,labElse);
        generateExpr((Object[])e[2]);
        emit("(Go␣_"+labEnd+")");
        emit("_"+labElse+":");
        generateExpr((Object[])e[3]);
        emit("_"+labEnd+":");
        return;
    case "CALL":
        // e = {"CALL",name,args}
        Object[] args = (Object[])e[2];
        int i;
        for( i=0 ; i!=args.length ; i++ )
            if( i==0 )
```

```
                generateExpr((Object[])args[i]);
            else
                generateExprP((Object[])args[i]);
        emit("(Call␣#\""+e[1]+"[f"+i+"]\"␣"+i+")");
        return;
    default:
        throw new Error("Unknown␣intermediate␣code␣type:␣\""+(String)e[0]+"\"");
    }
}

// Notkun: generateJump(e,labTrue,labTrue);
// Fyrir:  e er milliþula fyrir segð, labTrue og
//         labFalse eru heiltölur sem standa fyrir
//         merki eða eru núll.
// Eftir:  Búið er að skrifa lokaþulu fyrir segðina
//         á aðalúttak.  Lokaþulan veldur stökki til
//         merkisins labTrue ef segðina skilar sönnu,
//         annars stökki til labFalse.  Ef annað merkið
//         er núll þá er það jafngilt merki sem er rétt
//         fyrir aftan þulu segðarinnar.
static void generateJump( Object[] e, int labTrue, int labFalse )
{
    switch( (String)e[0] )
    {
    case "LITERAL":
        String literal = (String)e[1];
        if( literal.equals("false") || literal.equals("null") )
        {
            if( labFalse!=0 ) emit("(Go␣_"+labFalse+")");
            return;
        }
        if( labTrue!=0 ) emit("(Go␣_"+labTrue+")");
        return;
    default:
        generateExpr(e);
        if( labTrue!=0 ) emit("(GoTrue␣_"+labTrue+")");
        if( labFalse!=0 ) emit("(GoFalse␣_"+labFalse+")");
```

```
    }
}

// Notkun: generateJumpP(e,labTrue,labFalse);
// Fyrir:  e er milliþula fyrir segð, labTrue og
//         labFalse eru heiltölur sem standa fyrir
//         merki eða eru núll.
// Eftir:  Þetta kall býr til lokaþulu sem er jafngild
//         þulunni sem köllin
//            emit("(Push)");
//            generateJump(e,labTrue,labFalse);
//         framleiða.  Þulan er samt ekki endilega sú
//         sama og þessi köll framleiða því tilgangurinn
//         er að geta framleitt betri þulu.
static void generateJumpP( Object[] e, int labTrue, int labFalse )
{
    switch( (String)e[0] )
    {
    case "LITERAL":
        String literal = (String)e[1];
        emit("(Push)");
        if( literal.equals("false") || literal.equals("null") )
        {
            if( labFalse!=0 ) emit("(Go␣"+labFalse+")");
            return;
        }
        if( labTrue!=0 ) emit("(Go␣"+labTrue+")");
        return;
    default:
        generateExprP(e);
        if( labTrue!=0 ) emit("(GoTrue␣"+labTrue+")");
        if( labFalse!=0 ) emit("(GoFalse␣"+labFalse+")");
    }
}

// Notkun: generateExprR(e);
// Fyrir:  e er milliþula fyrir segð.
```

```
// Eftir:  Þetta kall býr til lokaþulu sem er jafngild
//         þulunni sem köllin
//            generateExpr(e);
//            emit("(Return)");
//         framleiða.  Þulan er samt ekki endilega sú
//         sama og þessi köll framleiða því tilgangurinn
//         er að geta framleitt betri þulu.
static void generateExprR( Object[] e )
{
    switch( (String)e[0] )
    {
    case "FETCH":
        // e = {"FETCH",pos}
        emit("(FetchR␣"+e[1]+")");
        return;
    case "LITERAL":
        // e = {LITERAL,literal}
        emit("(MakeValR␣"+(String)e[1]+")");
        return;
    case "IF":
        // e = {IF,cond,then,else}
        int labElse = newLab();
        generateJump((Object[])e[1],0,labElse);
        generateExprR((Object[])e[2]);
        emit("_"+labElse+":");
        generateExprR((Object[])e[3]);
        return;
    case "CALL":
        // e = {CALL,name,args}
        Object[] args = (Object[])e[2];
        int i;
        for( i=0 ; i!=args.length ; i++ )
            if( i==0 )
                generateExpr((Object[])args[i]);
            else
                generateExprP((Object[])args[i]);
        emit("(CallR␣#\""+e[1]+"[f"+i+"]\"␣"+i+")");
```

```
            return;
        default:
            generateExpr(e);
            emit("(Return)");
            return;
        }
    }

    // Notkun: generateExprP(e);
    // Fyrir:  e er milliþula fyrir segð.
    // Eftir:  Þetta kall býr til lokaþulu sem er jafngild
    //         þulunni sem köllin
    //             emit("(Push)");
    //             generateExpr(e);
    //         framleiða.  Þulan er samt ekki endilega sú
    //         sama og þessi köll framleiða því tilgangurinn
    //         er að geta framleitt betri þulu.
    static void generateExprP( Object[] e )
    {
        switch( (String)e[0] )
        {
        case "FETCH":
            // e = {"FETCH",pos}
            emit("(FetchP␣"+e[1]+")");
            return;
        case "LITERAL":
            // e = {"LITERAL",literal}
            emit("(MakeValP␣"+(String)e[1]+")");
            return;
        case "IF":
            // e = {"IF",cond,then,else}
            int labElse = newLab();
            int labEnd = newLab();
            generateJumpP((Object[])e[1],0,labElse);
            generateExpr((Object[])e[2]);
            emit("(Go␣␣"+labEnd+")");
            emit("_"+labElse+":");
```

```
            generateExpr((Object[])e[3]);                                                                    593
            emit("_"+labEnd+":");                                                                            594
            return;                                                                                          595
        case "CALL":                                                                                         596
            // e = {"CALL",name,args}                                                                        597
            Object[] args = (Object[])e[2];                                                                  598
            int i;                                                                                           599
            for( i=0 ; i!=args.length ; i++ ) generateExprP((Object[])args[i]);                              600
            if( i==0 ) emit("(Push)");                                                                       601
            emit("(Call␣#\""+e[1]+"[f"+i+"]\"␣"+i+")");                                                      602
            return;                                                                                          603
        }                                                                                                    604
    }                                                                                                        605
                                                                                                             606
    // Notkun (af skipanalínu):                                                                              607
    //        java NanoLisp forrit.s > forrit.masm                                                          608
    // Fyrir: Skráin forrit.s inniheldur löglegt NanoLisp                                                    609
    //        forit.                                                                                        610
    // Eftir: Búið er að þýða forritið og skrifa lokaþuluna                                                  611
    //        í skrána forrit.masm.  Sé sú lokaþula þýdd með                                                612
    //        skipuninni                                                                                    613
    //           morpho -c forrit.masm                                                                      614
    //        þá verður til keyrsluhæfa Morpho skráin forrit.mexe.                                          615
    public static void main( String[] args )                                                                 616
        throws IOException                                                                                   617
    {                                                                                                        618
        Lexer lexer = new Lexer(new FileReader(args[0]));                                                    619
        String name = args[0].substring(0,args[0].lastIndexOf('.'));                                         620
        NanoLisp parser = new NanoLisp(lexer);                                                               621
        Object[] intermediate = parser.program();                                                            622
        if( lexer.getToken()!='$' ) throw new Error("Expected␣EOF,␣found␣"+lexer.getLexeme());               623
        generateProgram(name,intermediate);                                                                  624
    }                                                                                                        625
}                                                                                                            626
```

Lokaþulusmiðurinn samanstendur af klasaboðunum generateProgram, generateFunction, generateExpr, generateExprR, generateExprP, generateJump og generateJumpP. Úttakið úr þáttaranum er sent í generateProgram sem sér um að skrifa lokaþulu fyrir forritið sem verið er að þýða. Sleppa mætti klasaboðunum generateExprP, generateExprR, generateJump og generateJumpP og útfæra aðeins generateExpr á eftirfarandi hátt.

```
// Notkun: generateExpr(e);
// Fyrir:  e er milliþula fyrir segð.
// Eftir:  Búið er að skrifa lokaþulu fyrir segðina
//         á aðalúttak.  Lokaþulan reiknar gildi
//         segðarinnar og skilur gildið eftir í
//         gildinu ac.
static void generateExpr( Object[] e )
{
    switch( (CodeType)e[0] )
    {
    case NAME:
        // e = {"NAME",pos}
        emit("(Fetch "+e[1]+")");
        return;
    case LITERAL:
        // e = {"LITERAL",literal}
        emit("(MakeVal "+(String)e[1]+")");
        return;
    case IF:
        // e = {"IF",cond,then,else}
        int labElse = newLab();
        int labEnd = newLab();
        generateExpr((Object[])e[1]);
        emit("(GoFalse _"+labElse+")");
        generateExpr((Object[])e[2]);
        emit("(Go _"+labEnd+")");
        emit("_"+labElse+":");
        generateExpr((Object[])e[3]);
        emit("_"+labEnd+":");
        return;
    case CALL:
        // e = {"CALL",name,args}
        Object[] args = (Object[])e[2];
        if( args.length!=0 )
            generateExpr((Object[])args[0]);
        for( int i=1 ; i<args.length ; i++ )
        {
```

```
                emit("(Push)");
                generateExpr((Object[])args[i]);
            }
            emit("(Call␣#\""+e[1]+"[f"+args.length+"]\"␣"+args.length+")");
            return;
        }
    }
```

Þulan yrði þá lengri og hægvirkari og halaendurkvæm föll yrðu þá ekki sjálfkrafa að lykkju eins og gerist þegar skipunin CallR er notuð. En þýðandinn myndi þá styttast þó nokkuð.

12.2 Þýðandi með tólunum JFlex og BYACC/J

Flestir þýðendur eru smíðaðir með hjálp tóla sem hönnuð eru til þýðandasmíðar. Annars vegar nota menn forrit sem hjálpa til að smíða lesgreina (*lexical analyser, lexer, scanner*), eitt slíkt tól er JFlex[2]. Hins vegar nota menn forrit sem hjálpa til að smíða þáttara (*parser, syntax analyser*), eitt slíkt er BYACC/J[3]. Bæði þessi tól eru ætluð til notkunar með Java forritunarmálinu. Einnig eru til svipaðir lesgreinasmiðir og þáttarasmiðir fyrir önnur forritunarmál svo sem C, C++, Python, Haskell og mörg fleiri.

12.2.1 Lesgreinir

Eftirfarandi lesgreinir er skrifaður í því máli sem JFlex lesgreinasmiðurinn skilur. Lesgreinirinn er í skránni `nanolisp.jflex`. JFlex les þessa skrá og skrifar Java forritstexta fyrir lesgreininn.

[2] http://jflex.de/
[3] http://byaccj.sourceforge.net/

```
/**
    JFlex lesgreinir fyrir NanoLisp.
    Höfundur: Snorri Agnarsson, janúar 2014

    Þennan lesgreini má þýða með skipununum
        java -jar JFlex.jar nanolisp.jflex
        javac NanoLispLexer.java
 */

%%

%public
%class NanoLispLexer
%unicode
%byaccj

%{

public NanoLispParser yyparser;

public NanoLispLexer( java.io.Reader r, NanoLispParser yyparser )
{
    this(r);
    this.yyparser = yyparser;
}

%}

  /* Reglulegar skilgreiningar */

  /* Regular definitions */

_DIGIT=[0-9]
_FLOAT={_DIGIT}+\.{_DIGIT}+([eE][+-]?{_DIGIT}+)?
_INT={_DIGIT}+
_STRING=\"([^\"\\]|\\b|\\t|\\n|\\f|\\r|\\\"|\\\'|\\\\|(\\[0-3][0-7][0-7])|\\[0-7][0-7]|\\[0-7])*\"
_CHAR=\'([^\'\\]|\\b|\\t|\\n|\\f|\\r|\\\"|\\\'|\\\\|(\\[0-3][0-7][0-7])|(\\[0-7][0-7])|(\\[0-7]))\'
```

```
_DELIM=[()]
_NAME=([:letter:]|[\+\-*/!%&=><\:\^\~&|?]|{_DIGIT})+

%%

{_DELIM} {
    yyparser.yylval = new NanoLispParserVal(yytext());
    return yycharat(0);
}

{_STRING} | {_FLOAT} | {_CHAR} | {_INT} | null | true | false {
    yyparser.yylval = new NanoLispParserVal(yytext());
    return NanoLispParser.LITERAL;
}

"if" {
    return NanoLispParser.IF;
}

"define" {
    return NanoLispParser.DEFINE;
}

{_NAME} {
    yyparser.yylval = new NanoLispParserVal(yytext());
    return NanoLispParser.NAME;
}

";".*$ {
}

[ \t\r\n\f] {
}

. {
    return NanoLispParser.YYERRCODE;
}
```

12.2.2 Þáttari og þulusmiður

Eftirfarandi þáttari er skrifaður í því máli sem BYACC/J þáttarasmiðurinn skilur. Þáttarinn er í skránni `nanolisp.byacc`. Þáttarasmiðurinn BYACC/J les þessa skrá og skrifar Java forritstexta fyrir þáttarann. Þáttarinn er einnig milliþulusmiður (*intermediate code generator*) og í sömu skrá er forritstextinn fyrir sama lokaþulusmið (*final code generator*) og notaður er í skránni `NanoLisp.java`, sem einnig er notaður í þessum þýðanda.

Línur 22 til 57 skilgreina málfræðina sem þáttað er eftir, ásamt smíð milliþulunnar. Takið eftir að málfræðin er skilgreind á sniði sem er nauðalíkt BNF. Þáttarasmiðir eins og BYACC/J vinna úr slíkri mállýsingu og búa til þáttara sem ber kennsl á málið sem mállýsingin lýsir.

Línur 120 til 266 skilgreina lokaþulusmiðinn sem (eins og í `NanoLisp.java`) samanstendur af föllunum generateProgram, generateFunction, generateExpr, generateExprP og generateExprR ásamt hjálparföllum.

```
/*
 * Byacc/J þáttari fyrir NanoLisp.
 * Höfundur: Snorri Agnarsson, 2014-2016.
 *
 * Þáttara þennan má þýða með skipununum
 *    byacc -J -Jclass=NanoLispParser nanolisp.byaccj
 *    javac NanoLispParser.java NanoLispParserVal.java
 */

%{
    import java.io.*;
    import java.util.*;
%}

%token <sval> LITERAL,NAME
%token IF,DEFINE
%type <obj> program, fundecl, expr, args
%type <ival> ids

%%

start
    :   program { generateProgram(name,((Vector<Object>)($1)).toArray()); }
    ;

program
    :   program fundecl     { ((Vector<Object>)($1)).add($2); $$=$1; }
    |   fundecl             { $$=new Vector<Object>(); ((Vector<Object>)($$)).add($1); }
    ;

fundecl
    :       {
                varCount = 0;
                varTable = new HashMap<String,Integer>();
            }
        '(' DEFINE '(' NAME ids ')' expr ')'
            {
```

```
                $$=new Object[]{$5,$6,$8};
            }
    ;

ids
    :   /* empty */             { $$=0; }
    |   ids NAME                { addVar($2); $$=$1+1; }
    ;

expr
    :   NAME                        { $$=new Object[]{"FETCH",findVar($1)}; }
    |   LITERAL                     { $$=new Object[]{"LITERAL",$1}; }
    |   '(' IF expr expr expr ')'   { $$=new Object[]{"IF",$3,$4,$5}; }
    |   '(' NAME args ')'           { $$=new Object[]{"CALL",$2,((Vector<Object>)($3)).toArray()}; }
    ;

args
    :   /* empty */         { $$=new Vector<Object>(); }
    |   args expr           { ((Vector<Object>)($1)).add($2); $$=$1; }
    ;

%%

    static private String name;
    private NanoLispLexer lexer;
    private int varCount;
    private HashMap<String,Integer> varTable;

    private void addVar( String name )
    {
        if( varTable.get(name) != null )
            yyerror("Variable "+name+" already exists");
        varTable.put(name,varCount++);
    }

    private int findVar( String name )
    {
```

```
    Integer res = varTable.get(name);                                                    75
    if( res == null )                                                                    76
        yyerror("Variable "+name+" does not exist");                                     77
    return res;                                                                          78
}                                                                                        79
                                                                                         80
int last_token_read;                                                                     81
                                                                                         82
private int yylex()                                                                      83
{                                                                                        84
    int yyl_return = -1;                                                                 85
    try                                                                                  86
    {                                                                                    87
        yylval = null;                                                                   88
        last_token_read = yyl_return = lexer.yylex();                                    89
        if( yylval==null )                                                               90
            yylval = new NanoLispParserVal(NanoLispParser.yyname[yyl_return]);           91
    }                                                                                    92
    catch (IOException e)                                                                93
    {                                                                                    94
        System.err.println("IO error: "+e);                                           95
    }                                                                                    96
    return yyl_return;                                                                   97
}                                                                                        98
                                                                                         99
public void yyerror( String error )                                                     100
{                                                                                       101
    System.out.println("Error:  "+error);                                            102
    System.out.println("Token:  "+NanoLispParser.yyname[last_token_read]);            103
    System.exit(1);                                                                     104
}                                                                                       105
                                                                                        106
public NanoLispParser( Reader r )                                                       107
{                                                                                       108
    lexer = new NanoLispLexer(r,this);                                                  109
}                                                                                       110
                                                                                        111
```

```
public static void main( String args[] )
    throws IOException
{
    NanoLispParser yyparser = new NanoLispParser(new FileReader(args[0]));
    name = args[0].substring(0,args[0].lastIndexOf('.'));
    yyparser.yyparse();
}

public static void emit( String s )
{
    System.out.println(s);
}

static void generateProgram( String name, Object[] p )
{
    emit("\""+name+".mexe\" = main in");
    emit("!{{");
    for( Object f: p ) generateFunction((Object[])f);
    emit("}}*BASIS;");
}

static void generateFunction( Object[] f )
{
    String fname = (String)f[0];
    int count = (Integer)f[1];
    emit("#\""+fname+"[f"+count+"]\" =");
    emit("[");
    generateExprR((Object[])f[2]);
    emit("];");
}

static int nextLab = 0;

static int newLab()
{
    return nextLab++;
}
```

```
static void generateExpr( Object[] e )
{
    switch( (String)e[0] )
    {
    case "FETCH":
        // e = {"FETCH",pos}
        emit("(Fetch␣"+e[1]+")");
        return;
    case "LITERAL":
        // e = {"LITERAL",literal}
        emit("(MakeVal␣"+(String)e[1]+")");
        return;
    case "IF":
        // e = {"IF",cond,then,else}
        generateExpr((Object[])e[1]);
        int labElse = newLab();
        int labEnd = newLab();
        emit("(GoFalse␣␣"+labElse+")");
        generateExpr((Object[])e[2]);
        emit("(Go␣␣"+labEnd+")");
        emit("_"+labElse+":");
        generateExpr((Object[])e[3]);
        emit("_"+labEnd+":");
        return;
    case "CALL":
        // e = {"CALL",name,args}
        Object[] args = (Object[])e[2];
        int i;
        for( i=0 ; i!=args.length ; i++ )
            if( i==0 )
                generateExpr((Object[])args[i]);
            else
                generateExprP((Object[])args[i]);
        emit("(Call␣#\""+e[1]+"[f"+i+"]\"␣"+i+")");
        return;
    default:
```

```
        throw new Error("Unknown intermediate code type: \""+(String)e[0]+"\"");
    }
}

static void generateExprR( Object[] e )
{
    switch( (String)e[0] )
    {
    case "FETCH":
        // e = {"FETCH",pos}
        emit("(FetchR "+e[1]+")");
        return;
    case "LITERAL":
        // e = {"LITERAL",literal}
        emit("(MakeValR "+(String)e[1]+")");
        return;
    case "IF":
        // e = {"IF",cond,then,else}
        generateExpr((Object[])e[1]);
        int labElse = newLab();
        emit("(GoFalse _"+labElse+")");
        generateExprR((Object[])e[2]);
        emit("_"+labElse+":");
        generateExprR((Object[])e[3]);
        return;
    case "CALL":
        // e = {"CALL",name,args}
        Object[] args = (Object[])e[2];
        int i;
        for( i=0 ; i!=args.length ; i++ )
            if( i==0 )
                generateExpr((Object[])args[i]);
            else
                generateExprP((Object[])args[i]);
        emit("(CallR #\""+e[1]+"[f"+i+"]\" "+i+")");
        return;
    default:
```

```
        generateExprR(e);                                                                    223
        emit("(Return)");                                                                    224
        return;                                                                              225
    }                                                                                        226
}                                                                                            227
                                                                                             228
static void generateExprP( Object[] e )                                                      229
{                                                                                            230
    switch( (String)e[0] )                                                                   231
    {                                                                                        232
    case "FETCH":                                                                            233
        // e = {"FETCH",pos}                                                                 234
        emit("(FetchP␣"+e[1]+")");                                                           235
        return;                                                                              236
    case "LITERAL":                                                                          237
        // e = {"LITERAL",literal}                                                           238
        emit("(MakeValP␣"+(String)e[1]+")");                                                 239
        return;                                                                              240
    case "IF":                                                                               241
        // e = {"IF",cond,then,else}                                                         242
        generateExprP((Object[])e[1]);                                                       243
        int labElse = newLab();                                                              244
        int labEnd = newLab();                                                               245
        emit("(GoFalse␣␣"+labElse+")");                                                      246
        generateExpr((Object[])e[2]);                                                        247
        emit("(Go␣␣"+labEnd+")");                                                            248
        emit("_"+labElse+":");                                                               249
        generateExpr((Object[])e[3]);                                                        250
        emit("_"+labEnd+":");                                                                251
        return;                                                                              252
    case "CALL":                                                                             253
        // e = {"CALL",name,args}                                                            254
        Object[] args = (Object[])e[2];                                                      255
        int i;                                                                               256
        for( Object arg: args ) generateExprP((Object[])arg);                                257
        if( args.length==0 ) emit("(Push)");                                                 258
        emit("(Call␣#\""+e[1]+"[f"+args.length+"]\"␣"+args.length+")");                      259
```

```
            return;                                                    260
        default:                                                       261
            emit("(Push)");                                            262
            generateExprR(e);                                          263
            return;                                                    264
        }                                                              265
    }                                                                  266
```

13 NanoLisp notkunardæmi

Eftirfarandi forritstexti, í skránni test.s, er dæmi um löglegt forrit í NanoLisp.

```
; MiniLisp prófunarforrit.
; Höfundur: Snorri Agnarsson, janúar 2014.

; Notkun: (fibo n)
; Fyrir:  n er heiltala, n>=0
; Gildi:  n-ta Fibonacci talan
(define (fibo n)
  (if (< n 2)
      1
      (+ (fibo (- n 1)) (fibo (- n 2)))
  )
)

; Notkun: (main)
; Fyrir:  Ekkert
; Eftir:  Búið er að reikna og skrifa fibo(30)
(define (main)
  (writeln (++ "fibo(30)=" (fibo 30)))
)
```

Ef við keyrum NanoLisp þýðandann með skipuninni

```
java NanoLisp test.s
```

eða hinn þýðandann með skipuninni

```
java NanoLispParser test.s
```

fáum við eftirfarandi úttak, sem er Morpho smalamál.

```
" test .mexe" = main in
!{{
#"fibo[f1]" =
[
(Fetch 0)
(MakeValP 2)
(Call #"<[f2]" 2)
(GoFalse _1)
(MakeValR 1)
_1:
(Fetch 0)
(MakeValP 1)
(Call #"-[f2]" 2)
```

```
(Call #"fibo[f1]" 1)                                    14
(FetchP 0)                                              15
(MakeValP 2)                                            16
(Call #"−[f2]" 2)                                       17
(Call #"fibo[f1]" 1)                                    18
(CallR #"+[f2]" 2)                                      19
];                                                      20
#"main[f0]" =                                           21
[                                                       22
(MakeVal "fibo(30)=")                                   23
(MakeValP 30)                                           24
(Call #"fibo[f1]" 1)                                    25
(Call #"++[f2]" 2)                                      26
(CallR #"writeln[f1]" 1)                                27
];                                                      28
}}*BASIS;                                               29
```

Þetta úttak er löglegt inntak í Morpho þýðandann, sem getur þýtt Morpho smalamál. Ef þetta úttak er í skránni `test.masm` má þýða það með skipuninni

```
morpho -c test.masm
```

eða með skipuninni

```
java -jar morpho.jar -c test.masm
```

Síðan má keyra forritið sem út kemur (í skrána `test.mexe`) með skipuninni

```
morpho test
```

eða

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

Þegar forritið er keyrt skrifast út línan

```
fibo(30)=1346269
```

Einnig má nota skipanarununa

```
java NanoLisp test.s | morpho -c
morpho test
```

til að þýða og keyra.

Athugið að til þess að þessar skipanir virki þarf skipunin `morpho` að vera í `PATH` og skráin `morpho.jar` þarf að vera í núverandi möppu (*current directory*).

Skrárnar `NanoLisp.java`, `morpho.jar`, `test.s` og aðrar skrár fyrir NanoLisp verða aðgengilegar í Uglunni fyrir nemendur í TÖL202M.