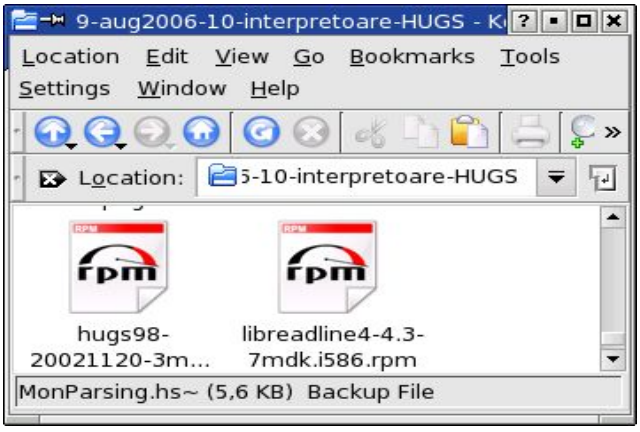
C1

Practica interpretării monadice

Despre: Un prim back-end de interpretor; Componentele acestuia; Extensibilitatea și adaptabilitatea lor

Această expunere se doreste a fi pragmatică. Presupun din start că aveți la dispoziție interpretorul Hugs al limbajului Haskell. Versiunea pe care am folosit-o este livrată împreună cu distributia Mandarke Linux 10.0, distribuție de download pe care o puteți descărca și folosi fără restricții. Și alte distribuții din aceeași perioadă pot folosi interpretorul. L-am testat de asemenea folosind Suse Linux 9.3.



Cele două pachete .rpm necesare

Ceea ce aveți de făcut pentru a-l instala pe o altă distribuție este să copiați fișierele din imagine de pe CD-urile sau DVD-ul Mandrake 10.0 și să instalați cele două pachete: interpretorul HUGS de pe CD-ul al III-lea și biblioteca libreadline-4-4.3-7mdk de pe CD-ul numărul dintâi.

Compilatorul de Haskell GHC, mai modern, este de asemenea o opțiune de luat în considerare, dar deoarece limbajul Haskell este încă în dezvoltare e posibil sau nu să mai existe mici incompatibilități.

Experimentul 0:

Scrieți un back-end de interpretor pentru un mic While-language. (Sursa de mai jos conține explicații. Voi prezenta de asemenea o serie de comentarii în continuarea ei.)

```
------------------------------------------------------------
-- Implementarea back-end-ului unui While Language: DSL.hs
-- dupa versiunea in metaML din lucrarea
-- "DSL Implementation using staging and monads"
-- de Tim Sheared, Zine-el-abidine Benaisa, Emir Pasalic
-- Pacific Software Research Center, Oregon Graduate Institute
-- (lucrarea este nedatata)
-- implementat in Haskell de Dan Popa
-- 1-6 iulie 2006, refacut la 1 august 2006
------------------------------------------------------------
-- Sintaxa limbajului
--- Exemplu de program
--
--  Exemplu de sintaxa concreta a unui program in limbaj While
-- declare x = 150 in
--   declare y = 200 in
--     { while  x > 0 do { x:=x-1; y:=y-1};
--       print y}
-- Limbajul (prin exemple)
-- Atribuirea de intregi:                  x:=1
-- Secventa de doua instructiuni:          { x:=1; y:=2; }
-- Decizia/Alternativa/Conditionala:       if x then x:=1 else  y:=1
-- Bucla while:                                  while x > 0 do x:= x-1
-- Declaratia locala:                      declare x=1 in x:=x+1
-- Print, afisarea expresiei:              print x
```

```
-----------------------------------------------
import Prelude hiding (read)

-- Declaratiile tipului de date pentru expresii
-- Exemple de expresii (vor fi necesare si in frontend, la scrierea parserului )
data Exp =  Constant Int      -- 5
         | Variable String   -- x
         | Minus Exp Exp     -- x - 5
         | Greater Exp Exp   -- x > 1
         | Times Exp Exp     -- X * 5
         deriving Show

-- Declaratiile tipului de date pentru comenzi
data Com =  Assign String Exp
         | Seq Com Com             -- { x:=1; y:=2; }
         | Cond Exp Com Com        -- if x then x:=1 else  y:=1
         | While Exp Com           -- while x > 0 do x:= x-1
         | Declare String Exp Com  -- declare x=1 in x:=x+1
         | Print Exp               -- print x
         deriving Show

-- Exemplu de sintaxa concreta a unui program in limbaj While
--  declare x = 150 in
--    declare y = 200 in
--      {while x > 0 do { x:=x-1; y:=y-1 };
--       print y }

-----------------------------------------------
-- Exemplul de sursa de program de mai sus, in sintaxa abstracta,
-- asa cum ar rezulta dupa terminarea parsarii. (arbore)

s1 = Declare "x" (Constant 150)
     (Declare "y" (Constant 200)
       (Seq (While (Greater (Variable "x" ) (Constant 0)
                )
                (Seq (Assign "x" (Minus (Variable "x")
                                        (Constant 1)
                                 )
                     )
                     (Assign "y" (Minus (Variable "y")
                                        (Constant 1)
                                 )
                     )
                )
            )
            (Print (Variable "y"))
       )
     )

-- Monada inlocuieste, simuleaza masina virtuala cu "environment"
-- (aici doar o lista de nume de variabile) si stiva
```

```
type Location = Int
type Index = [String]
type Stack = [Int]
                        -- Stiva (se pot extinde tipurile
                        -- de date )

-------------------------------------------
~ Functii necesare manipularii elementelor din context
-

-- 1) functia care da numarul/pozitia unui nume in lista context
position          :: String  -> Index ->  Location   -- locatia era doar un numar
position name index = let
                  pos n (nm:nms) = if name == nm
                                   then n
                                   else pos (n+1) nms
                                   -- param n creste ,lista scade
                in pos 1 index        -- de la nr 1  plecam cu cautarea

-- 2) functia care extrage de pe stiva intregul din pozitia indicata
-- acest intreg este valoarea variabilei
fetch          :: Location -> Stack -> Int
fetch n (v:vs)   =  if n == 1 then v else fetch (n-1) vs

~ Nota: nu-i prevazut cazul cand n este mai mare ca lungimea stivei
-- deoarece in realitate nu se poate ajunge la asa ceva.

-- 3) functia care pune ... in stiva, operand ca intr-un vector !!
put          :: Location -> Int -> Stack -> Stack
put n x (v:vs) = if n==1     -- daca se cerea punere in pozitia n==1
             then x:vs  -- rezulta aceeasi stiva dar cu un nou cap/varf
             else v:(put (n-1) x vs)

~ Monada M a Starilor si a Iesirilor (STack and OUTput)
-- incapsuleaza intr-un tip construit cu un constructor numit StOut
-- functia de trecere de la o stiva Stack la o tripleta (a, Stack, String)
-- unde a este valoarea polimorfica prelucrata de monada (incapsulata)
-- iar string este stringul de iesire.

newtype M a = StOut (Stack -> (a, Stack, String))

instance Monad M where
  return x = StOut (\n -> (x,n, ""))
  e >>= f  = StOut (\n -> let (a,n1,s1) = (unStOut e) n
                          (b,n2,s2) = unStOut (f a) n1
                      in (b,n2,s1++s2) )
-- unde
unStOut (StOut f) = f

-- Operatii curente ce dau valori monadice, vor fi folosite in interpretor
~ un fetch adus in monada
```

```
-- aduce din stiva o valoare spre prelucrare
getfrom :: Location -> M Int
getfrom i = StOut (\ns -> (fetch i ns, ns, ""))

-- modifica stiva punind in ea o valoare, intr-o anume pozitie (ca la vector)
write :: Location -> Int -> M ()
write i v = StOut (\ns -> ( (), put i v ns, "") )

-- modifica stiva punind in ea o valoare, in cap / top
push :: Int -> M ()
push x = StOut(\ns -> ((), x:ns, "") )

-- acest pop este un fel de "return_pop " si este folosit ca atare
pop :: M ()
pop = StOut (\m -> let
                    (n:ns) = m
                  in
                    ( () , ns ,"" )
            )
-~ e necesar deoarece nu puteam folosi la finalul unor definitii in do-notatie
-- return (). Prin felul cum e definit return incapsuleaza o functie care nu
-- modifica stiva, deci nu are efectul lui pop


----------INTERPRETORUL MONADIC ----------
-- 1) - Evaluatorul de expresii aritmetice
-- evalueaza o expresie intr-un context si rezulta un intreg "incapsulat"

eval1          :: Exp -> Index -> M Int
eval1 exp index = case exp of
                  Constant n  -> return n
                  Variable x  -> let loc = position x index
                                 in getfrom loc
                  Minus x y   -> do { a <- eval1 x index ;
                                      b <- eval1 y index ;
                                      return (a-b) }
                  Greater x y -> do { a <- eval1 x index ;
                                      b <- eval1 y index ;
                                      return (if a > b
                                              then 1
                                              else 0) }
                  Times x y   -> do { a <- eval1 x index ;
                                      b <- eval1 y index ;
                                      return ( a * b ) }

-- Testarea valorii expresiei intregi
test a = unStOut (eval1 a [])  []

-- 2)  - Interpretorul de comenzi

interpret1 :: Com -> Index -> M ()
```

```
interpret1 stmt index = case stmt of
        Assign name e-> let loc = position name index
                in  do { v <- eval1 e index ;
                     write loc v }
        Seq s1 s2    -> do { x <- interpret1 s1 index ;
                     y <- interpret1 s2 index ;
                     return () }
        Cond e s1 s2 -> do { x <- eval1 e index ;
                     if x == 1
                       then interpret1 s1 index
                       else interpret1 s2 index }
        While e b    -> let loop () = do { v <- eval1 e index ;
                                  if v==0
                                   then return ()
                                   else
                                   do {interpret1 b index ;
                                      loop () } }
                in loop ()
        Declare nm e stmt ->  do { v <- eval1 e index ;
                           push v ;
                           interpret1 stmt (nm:index) ;
                           pop }
        Print e -> do { v <- eval1 e index ;
                output v}
```

– Testarea interpretorului

```
interp a = unStOut (interpret1 a []) []

output :: Show a => a -> M ()
output v = StOut (\n -> ((),n,show v))
```

Explicații / comentarii: acesta este back-end-ul unui interpretor de arbori sintactici, realizat cu două funcții de evaluare, una pentru evaluarea / interpretarea expresiilor întregi alta pentru evaluarea / interpretarea comenzilor.

Limbajul pentru care este realizat acest back-end de interpretor e o variantă de While-language dotată cu: declarați e de variabilă , secvență de exact două instrucțiuni, buclă while, atribuire, expresii întregi, if și print. Expresiile conțin constante, variabile, comparația cu “>” și înmulțirea. (Introducerea celorlalte operații în limbaj se face în manieră similară.)

```
–  Exemplu de sintaxa concreta a unui program in limbaj While
– declare x = 150 in
–   declare y = 200 in
~     { while  x > 0 do { x:=x-1; y:=y-1};
–       print y}

– Limbajul (prin exemple)
– Atribuirea de intregi:                x:=1
– Secventa de doua instructiuni:        { x:=1; y:=2; }
– Decizia/Alternativa/Conditionala:     if x then x:=1 else  y:=1
– Bucla while:                                  while x > 0 do x:= x-1
– Declaratia locala:                    declare x=1 in x:=x+1
– Print, afisarea expresiei:            print x
```

Notați și faptul că această sintaxă este orientativă. Aici este dată doar pentru a vă face o idee asupra limbajului. În realitate back-end-ul interpretorului prelucrează arbori sintactici, deci nu are de-a face cu sintaxa concretă de mai sus. Pentru o înmulțire a două expresii el va procesa un arbore alcătuit din expresia stângă (pe post de subarbore stâng), expresia dreaptă (pe post de subarbore drept) și nodul lor părinte marcat cu numele operației (în exemplu nostr acesta fiind “Times”).

Arborii sintaxei abstracte (AST) . Declarăm două feluri de arbori, numiți “Exp” și “Com”. Sunt tipuril de arbori pentru reprezentat expresii și tipul de arbori pentru reprezentat comenzi. (Notă: Aceste declarații se vor folosi ulterior și la front-end-ul interpretorului, ele fiind practic interfața între front-end și back-end așa că scrieți-le cu simț de răspundere, cu nume sugestive, clare.) Puteți schimba aceste nume (“Exp” și “Com”) deoarece în Haskell constructorii de tipuri utilizator sunt funcții care pot avea orice nume doriți. În dialectul Haskell recunoascut de interpretorul Hugs tipurile de arbori se introduc cu simple declarații **data. (data – scrisă cu minuscule -** este declarația Haskell care introduce noi tipuri de date). Bara verticală semnifică reuniunea de tipuri.

```
– Declaratiile tipului de date pentru expresii
data Exp =  Constant Int       – 5
        | Variable String   – x
        | Minus Exp Exp      – x - 5
        | Greater Exp Exp    – x > 1
        | Times Exp Exp      – X * 5
        deriving Show
```

"Constant", "Variable", "Minus" , "Greater" și "Times" se numesc în Haskell constructori de date. Numele nu sunt rezervate, dar trebuie să înceapă cu majusculă. În practică ei corespund cu etichetele nodurilor operatori din arborele operatorial.

Tipul arborilor expresiilor este un tip recursiv (apare "Exp" în ambele părți ale egalității) iar variantele de noduri rădăcină sunt etichetate cu "Constant", "Variable", "Minus" , "Greater" și "Times".

Declarația *deriving Show* introduce tipul nou creat în clasa tipurilor afișabile. Efectul este că Hugs va putea tipări valorile noului tip (într-un format implicit dedus din formatele pentru subtipurile sale) fără să mai scriem noi în mod special o funcție de afișare a arborilor. Int este un tip întreg din Haskell.

Exercițiu de adaptabilitate: Schimbați aceste nume de operatori cu altele, de ce nu, traduse sau scrise în altă limbă. Astfel, prin simpla schimbare de nume în declarația **data** se obține adaptarea structurilor arborilor sintactici AST.

Tipul arborilor comenzilor se declară similar. El se numește în exemplul de mai sus "Com" . Sunt arbori care pot avea ca subarbori alți arbori "Com" sau arbori "Exp".

```
– Declaratiile tipului de date pentru comenzi
data Com =  Assign String Exp
        | Seq Com Com              – { x:=1; y:=2; }
        | Cond Exp Com Com         – if x then x:=1 else  y:=1
        | While Exp Com            – while x > 0 do x:= x-1
        | Declare String Exp Com   – declare x=1 in x:=x+1
        | Print Exp                – print x
       deriving Show
```

Tipul String este tip Haskell standard, predefinit și înseamnă simultan și string și listă de caractere. În Haskell listele de caractere coincid cu stringurile și există două notații: "1 dan" e echivalent cu lista formată din caracterele individuale ['1',' ','d','a','n'] și se pot aplica stringurilor operatorii de la liste. Constructorii de tipuri sunt și aici ale și ca să aibă nume semnificative: "Assign", "Seq", "Cond", "While", "Declare" și "Print". Ei corespund nodurilor etichetate cu aceleași denumiri din arborele operatorial al comenzilor și bineînțeles instrucț iunilor cu același nume.

Pentru testarea back-end-ului de interpretor vom folosi următorul program în While language:

```
~ Exemplu de sintaxa concreta a unui program in limbaj While
–  declare x = 150 in
~    declare y = 200 in
–      {while x > 0 do { x:=x-1; y:=y-1 };
–       print y }
```

Dar deoarece nu dispunem încă de front-end-ul interpretorului vom traduce artizanal acest text în arborele operatorial echivalent și vom memora ca o constantă acest arbore. Notat cu "s1", el are o dublă utilitate. Pe de o parte sugerează ce fel de arbori va produce front-end-ul interpretorul în urma analizei sintactice a textului sursă. Pe de altă parte el va fi exemplul de program cu care vom testa interpretorul. (Totodată el este și un bun exemplu de scriere a arborilor în Haskell, util pentru cititorii care nu cunosc limbajul).

```
_____
– Exemplul de sursa de program de mai sus, in sintaxa abstracta,
~ asa cum ar rezulta dupa terminarea parsarii. (arbore)

s1 = Declare "x" (Constant 150)
     (Declare "y" (Constant 200)
       (Seq (While (Greater (Variable "x" ) (Constant 0)
                  )
                  (Seq (Assign "x" (Minus (Variable "x")
```

```
                    (Constant 1)
                )
            )
            (Assign "y" (Minus (Variable "y")
                    (Constant 1)
                )
            )
        )
    )
    (Print (Variable "y"))
  )
)
```

Arborele sintaxei abstracte a programului de mai sus începe cu nodul rădăcină etichetat cu “Declare” , are un prim subarbore (cel stâng) de tip String cu numele variabilei “x” și încă doi subarbori (frați):

- arborele care corespunde expresiei constante 150
- și cel al instrucțiunii în care va fi valabilă declarația. Acesta este alt arbore care începe cu un nod etichetat “Declare”.

Mașina virtuală: Interpretoarele execută în cursul evaluării succesiuni de calcule. Acestea sunt realizate de “modelul de calcul”, un suport matematic al universului operațiilor. Adesea, se folosește o mașină virtuală, un mic procesor cu stivă (sau, alternativ cu regiștri) emulat prin soft. Poate avea și memorie (environment) separată sau poate stoca variabilele pe stivă (caz în care simpla lor poziție dată de un index e suficientă ca să le regăsim valorile).

Efectele laterale (side efects) ale execuției instrucțiunilor se reflectă asupra mașinii virtuale, o afectează, îi schimbă configurația de pe stivă și/sau din regiștri.

Declarațiile pentru numărul locației, lista denumirilor variabilelor

(numită uzual tabelă de simboluri) și stiva de valori întregi sunt :

```
type Location = Int
type Index = [String]
type Stack = [Int]
```

“Type” introduce în Haskell doar o sinonimie de tip ca și “Type” din Pascal, dând un nume nou tipului descris în dreapta semnului egal.

Exercițiu de adaptabilitate: Schimbați mulțimea valorilor care se pot stoca în stivă cu alt tip, de exemplu numere reale (Bool, Float, Char, String) apoi cu o reuniune de tipuri.

Manipularea stivei și indexului (tabelei de simboluri) se face mai ușor dacă dispunem de funcții în acest scop. Iată-le:

```
– 1) functia care da numarul/pozitia unui nume in lista context
position          :: String  -> Index ->  Location   -- locatia era doar un numar
position name index = let
                    pos n (nm:nms) = if name == nm
                                then n
                                else pos (n+1) nms
                in pos 1 index
```

Este o funcție tipică de extragere a valorii de pe o poziție dată dintr-o listă. La fiecare apel “pos” incrementează primul argument și caută elementul în sublista care începe cu acea poziție, transmisă ca al doilea argument. Căutarea începe la poziția 1, cu lista inițială dată.

```
– 2) functia care extrage de pe stiva intregul din pozitia indicata
~ acest intreg este valoarea variabilei
fetch          :: Location -> Stack -> Int
fetch n (v:vs)   =  if n == 1 then v else fetch (n-1) vs

– Nota: nu-i prevazut cazul cand n este mai mare ca lungimea stivei
– deoarece in realitate nu se poate ajunge la asa ceva.
```

Deoarece atribuirile vor modifica valoarea variabilelor trebuie să existe la nivelul modelului de calcul un mecanism de modificare a valorilor (care sunt pe stivă în cazul nostru). Funcția primește locația și valoarea și transformă stiva veche într-una nouă:

```
– 3) functia care pune ... in stiva, operand ca intr-un vector !!
put          :: Location -> Int -> Stack -> Stack
put n x (v:vs) = if n==1     -- daca se cerea punere in pozitia n==1
                 then x:vs  -- rezulta aceeasi stiva dar cu un nou cap/varf
                 else v:(put (n-1) x vs)
```

Dacă i se cere să o valoare care nu e în prima poziție funcția se apelează recursiv până găsește locația cerută.

Modelul de calcul mai cuprinde pe lîngă stivă și o ieșire, un șir de caractere care se lungește pe măsură ce se trimit noi rezultate la output cu *print*.

Deoarece semantica interpretorului intenționăm să se scrie în do-notație iar do-notația se bazează pe operațiile polimorfice “bind” (>>=) și “return” dintr-o monadă implementăm mașina virtuală printr-o monadă. Monada este, într-o viziune simplificată doar o structură algebrică formată - în Haskell – dintr-un constructor de tip polimorfic (un functor în termeni de teoria categoriilor) și două funcții polimorfice (două transformări naturale în termeni de teoria categoriilor) cu o anumită schemă de tip și care îndeplinesc niște axiome cunoscute drept legile monadei. Asemenea monade fiind studiate deja și implementate e suficient să alegem una: aici este folosită monada Stărilor și Ieșirilor .

```
– Monada M a Starilor si a Iesirilor (STack and OUTput)
– incapsuleaza intr-un tip construit cu un constructor numit StOut
– functia de trecere de la o stiva Stack la o tripleta (a, Stack, String)
– unde a este valoarea polimorfica prelucrata de monada (incapsulata)
```

```
– iar String este stringul de iesire.

newtype M a = StOut (Stack -> (a, Stack, String))

instance Monad M where

 return x = StOut (\n -> (x,n, ""))
 – x e valoarea incapsulata in element, pe stive actioneaza ca o functie identica
 – iar sirul de iesire,string-ul este output-ul ce se concateneaza, e vid


 e >>= f  = StOut (\n -> let (a,n1,s1) = (unStOut e) n
                         (b,n2,s2) = unStOut (f a) n1
                     in (b,n2,s1++s2) )
                          – functia scoasa din capsula se aplica lui n
                          – si rezulta o tripleta noua a, n1, s1 cuprinzand
                          – valoarea a,stiva noua n1
                          – a este apoi  prelucrata cu functia f (::a-> M b)
                          ~ rezulta o noua valoare monadica
                          –(functie ce prelucreaza stive) ce se aplica lui n1.
                          – se obtine stiva n2 si se concateneaza output-urile
```

Valorile din monadă sunt create cu constructorul StOut dintr-o funcție de la **Stack** la tripleta **(a, Stack, String)** care astfel este încapsulată. (În fond aceasta reflectă funcția de tranziție de configurații pentru mașina virtuală .) UnStOut scoate o asemenea funcție din capsulă pregătind-o de aplicare.

```
– unde
unStOut (StOut f) = f
```

Deoarece în biblioteca Standard Prelude este definită clasa monadelor (clasa Monad) introducerea noi monade se face cu o declarați *instance* după *declararea cu newtype a tipului elementelor monadei.* Cu ocazia declarației *instance* se precizează comportamentul celor doi operatori polimorfici, “return” și “bind” (>>=) în monada nou introdusă.

Return e funcția care transformă valori în valori monadice. (Nu

uitați că ele trebuie să aibă un tip comun și în același timp să reprezinte orice alt tip de valoare).

```
return x = StOut (\n -> (x,n, ""))
  – x e valoarea incapsulata in element, pe stive actioneaza ca o functie identica
  -- iar sirul de iesire,string-ul este output-ul ce se concateneaza e vid
```

Bind acționează pentru a combina o valoare monadică anterioară cu o funcție de la valori la valori monadice. În final se obține o altă funcție încapsulată (valoare monadică). Observați cum își prelucrează ea (funcția din capsulă) un eventual parametru n !

```
e >>= f  = StOut (\n -> let (a,n1,s1) = (unStOut e) n
                            (b,n2,s2) = unStOut (f a) n1
                        in (b,n2,s1++s2) )

                            -- functia scoasa din capsula e se aplica lui n
                            -- si rezulta o tripleta noua a, n1, s1 cuprinzand
                            -- valoarea a,stiva noua n1
                            -- a este apoi prelucrata cu functia f (::a-> M b)
                            -- rezulta o noua valoare monadica
                            --(functie ce prelucreaza stive) ce se aplica lui n1.
                            -- se obtine stiva n2 si se concateneaza output-urile
```

Operații curente: Deoarece interpretorul lucreaza cu valori monadice, funcțiile care lucrează cu stiva (și alte funcții necesare interpretorului) vor trebui să dea rezultatele sub formă de astfel de valori monadice, valori care încapsulează chiar funcțiile ce fac prelucrările de interes pentru noi (fetch, push, pop).

```
– aduce din stiva o valoare spre prelucrare
getfrom :: Location -> M Int
getfrom i = StOut (\ns -> (fetch i ns, ns, ""))
```

Funcția getfrom aduce sub formă de valoare monadică rezultatul căutării cu fetch în stivă.

```
-- modifica stiva punind in ea o valoare, intr-o anume pozitie (ca la vector)
```

```
write :: Location -> Int -> M ()
write i v = StOut (\ns -> ( (), put i v ns, "") )
```

Funcția write încapsulează într-o valoare monadică o prelucrare asupra stivei (punerea unui element cu “put”). Deoarece nu interesează rezultatul ci doar efectul asupra stivei, ca rezultat s-a pus tuplul vid () iar ca tip al variabilei polimorfice tipul acestuia, notat tot ().

```
– modifica stiva punind in ea o valoare, in cap / top
push :: Int -> M ()
push x = StOut(\ns -> ((), x:ns, "") )
```

Funcția push cu argumentul x încapsulează într-o valoare monadică prelucrarea asupra stivei ns care adaugă (cu “cons , notat :) la stiva ns elementul x.

```
– acest pop este un fel de ”return_pop ” si este folosit ca atare
pop :: M ()
pop = StOut (\m -> let
                    (n:ns) = m
                in
                  ( () , ns ,"" )
        )
```

Prelucrarea de stive încapsulată în rezultatul funcției pop este cea care duce stiva m = (n:ns) în stiva ns. Astfel se aruncă primul element.

Deși în general definițiile în do-notație ale regulilor semantice de interpretare se termină cu return este clar că există și situații, (cum e aceea în care se aruncă o variabilă la ieșirea din bloc) în care acest “return” așa cum a fost definit, nu poate fi folosit. Motivul: Acest “return” produce o valoare monadică în care prelucrarea stivei e făcută de funcția identică ce nu poate returna o stivă modificată.

Evaluatorul de expresii aritmetice: Evaluează o expresie Exp într-un context Index (stiva fiind ascunsă în "capsula" monadei) și returnează nu un Int ci un M Int, o valoare monadică. În general, dacă se evaluează valori ale altor tipuri, (Bool, Float, Char, String) se vor scrie funcțiile de evaluare corespunzătoare:

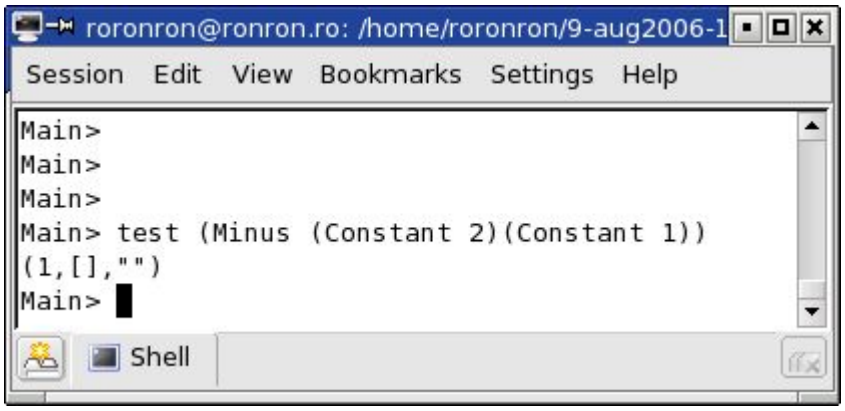
```
----------INTERPRETORUL MONADIC ----------
~ 1) - Evaluatorul de expresii aritmetice
-- evalueaza o expresie intr-un context si rezulta un intreg "incapsulat"

eval1       :: Exp -> Index -> M Int
eval1 exp index = case exp of
            Constant n  -> return n
            Variable x  -> let loc = position x index
                       in getfrom loc
            Minus x y   -> do { a <- eval1 x index ;
                          b <- eval1 y index ;
                          return (a-b) }
            Greater x y -> do { a <- eval1 x index ;
                          b <- eval1 y index ;
                          return (if a > b
                                 then 1
                                 else 0) }
            Times x y   -> do { a <- eval1 x index ;
                          b <- eval1 y index ;
                          return ( a * b ) }
```

Datorită existenței do-notației din Haskell scrierea formulelor compoziționale care descriu semantica limbajului, mai exact modul de evaluare al arborilor este imediată iar textul vorbește de la sine.

Testarea rezultatelor se face cu functia test. Functia unStOut scoate funcția încapsulată din valoarea monadică și o aplică listei vide. Rezultă o valoare care este afișabilă de către interpretor. (Valoarea funcțională în sine nu era.)

```
– Testarea valorii expresiei intregi
test a = unStOut (eval1 a []) []
```

Exemplu: Evaluarea expresiei 2 -1 scrisă ca arbore operatorial ne dă imediat rezultatul așteptat.



Se observă prima valoare. O serie de teste similare se fac și pentru celelalte forme posibile ale expresiilor. Demonstrația formală a funcționalității se poate face prin inducție structurală.

Exercițiu de adaptabilitate: Adăugați operatorii lipsă din expresiile cu numere întregi.

Interpretorul de comenzi:

```
– 2)  - Interpretorul de comenzi

interpret1 :: Com -> Index -> M ()
interpret1 stmt index = case stmt of
                Assign name e-> let loc = position name index
                          in  do { v <- eval1 e index ;
                                   write loc v }
                Seq s1 s2    -> do { x <- interpret1 s1 index ;
                                  y <- interpret1 s2 index ;
                                  return () }
                Cond e s1 s2 -> do { x <- eval1 e index ;
                                  if x == 1
```

```
                then interpret1 s1 index
                else interpret1 s2 index }
While e b   -> let loop () = do { v <- eval1 e index ;
                        if v==0
                        then return ()
                        else
                        do {interpret1 b index ;
                           loop () } }
          in loop ()
Declare nm e stmt ->  do { v <- eval1 e index ;
                    push v ;
                    interpret1 stmt (nm:index) ;
                    pop }
Print e -> do { v <- eval1 e index ;
          output v}

-- mai este nevoie de sa definim functia output
output :: Show a => a -> M ()
output v = StOut (\n -> ((),n,show v))
```

– Testarea interpretorului

```
interp a = unStOut (interpret1 a []) []
```

Atribuirea: Calculează poziția varaibilei invocate căutând numele ei în indexul de variabile. Apoi în același loc pune valoarea v aflată în urma evaluării expresiei e în contextul curent index.

```
Assign name e-> let loc = position name index
      in  do { v <- eval1 e index ;
           write loc v }
```

Secvența: Interpretează ambele expresii s1,s2, în contextul curent, notat tot “index”, de la momentul respectiv. Side-efect-ul asupra contextului e realizat, pe nevăzute, de operatorul “bind” (>>=) al monadei.

```
Seq s1 s2   -> do { x <- interpret1 s1 index ;
               y <- interpret1 s2 index ;
               return () }
```

Se observă că descrierea semanticii secvenței nu mai conține detalii legate de mecanismul de înlănțuire a efectelor asupra mașinii virtuale sau/și asupra environment-ului. Interpretorul este din acest motiv mai ușor adaptabil. Putem schimba (așa

cum face P.Wadler în [Wad-92abc]) monada de la baza interpretorului și obține alte interpretoare, de exemplu unul cu mesaje de eroare. Schimbarea se face fără a afecta descrierea semanticii de la acest nivel ceea ce dovedește superioritatea interpretoarelor monadice.

Alternativa: Evaluarea instrucțiunii alternative (condiționale) constă în evaluarea expresiei după care se ia decizia apoi în testarea valorii obținute. În interpretorul acesta, ca și în C, întregul 1 este considerat valoare booleană adevărată și determină execuția primei ramuri (s1). În caz contrar se execută structura (s2) de pe a doua ramură.

```
Cond e s1 s2 -> do { x <- eval1 e index ;
     if x == 1
     then interpret1 s1 index
     else interpret1 s2 index }
```

Bucla while: Autorii optează pentru o descriere recursivă a semanticii buclei while. Haskell permite structuri recursive cum ar fi liste infinite (pe care le stochează ca grafuri). Soluția este implementabilă și în alte limbaje funcționale (Ex: Meta ML).

```
While e b   -> let loop () = do { v <- eval1 e index ;
                  if v==0
                  then return ()
                  else
                     do {interpret1 b index ;
                        loop () } }
   in loop ()
```

Se remarcă evitarea apelului recursiv la întregului evaluator de comenzi, ceea ce contribuie la reducerea legăturii între constituienții interpretorului deci la adaptabilitatea acestuia. Faptul că instrucțiunea while va cicla rămâne o proprietate locală a implementării semanticii instrucțiunii și nu una a întregului evaluator.

Declarația de variabilă: Introduce valoarea “e” a variabilei “nm” în contextul curent. Se evaluează valoarea, se introduce variabila în stivă cu “push”, apoi se interpretează instrucțiunea următoare în contextul unui index de variabile care are pe prima poziție numele variabilei adăugate. (operatorul “:” este similar cu “cons” din LISP) .

```
Declare nm e stmt ->  do { v <- eval1 e index ;
        push v ;
        interpret1 stmt (nm:index) ;
        pop }
```

Ultima operație este “pop” (care returnează valoarea încapsulată monadic a tuplului vid) după ce a scos valoarea variabilei de pe stivă.

Instrucțiunea de ieșire “Print”.Întâi se evaluează expresia dată.

```
Print e -> do {         v <- eval1 e index ;
                        output v}
```

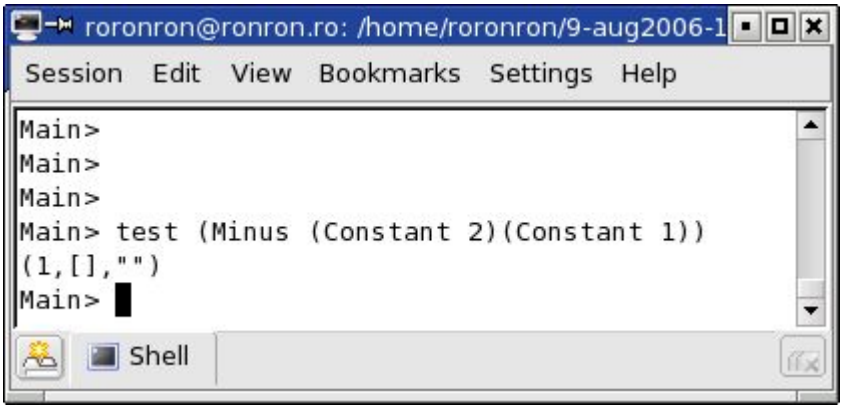
Aceasta e predată unei funcții polimorfice care produce o valoare monadică ce introduce în șirul de ieșire (al treilea parametru din tripletă) string-ul afișabil al valorii date, obținut cu funcția de sistem “show.” Deoarece show este polimorfică parametric și poate afișa toate valorile din tipurile de clasă Show și output va avea un parametru de tip a (nelegat) precum și precondiția ca acest a să fie de clasă Show. (Care se scrie Show a => ..... unde Show este clasa tipurilor Haskell cu valori afișabile).

```
output :: Show a => a -> M ()
output v = StOut (\n -> ((),n,show v))
```

Interpretarea arborelui programului s1**:**

```
declare x = 150 in
  declare y = 200 in
    {while x > 0 do { x:=x-1; y:=y-1 };
     print y }
```

duce la afișarea rezultatului așteptat, după 150 de decrementări.



Demonstrația matematică completă a funcționării unui back-end se realizează prin inducție structurală și constă în demonstrarea echivalenței semanticii rezultate dinaceastă interpretare cu o altă semantică a limbajului.

O testare testarea minimală (dar cel puțin o dată) a fiecărei părți a interpretorului se poate face prin furnizarea de exemple care să-i pretindă să evalueze fiecare dintre structurile recunoscute de acesta.

Pentru a putea introduce programe și în alt mod decât descriindu-le arborii operatoriali echivalenți, este nevoie și de prima parte a unui interpretor, cea care face *analiza sintactică* și *construcția arborelui sintaxei abstracte(AST)*, numită **front-end**.