```
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose  toHandle
          putStr  "Done."

getAndOpenFile          :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
       do putStr prompt
          name <- getLine
          catch (openFile name mode)
              (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                        getAndOpenFile prompt mode)
```

Din cauza modului de funcționare al sistemului de evaluare întârziată (eng. lazy evaluation) la execuția funcției **getContents**, conținutul fișierului nu este citit în memorie tot dintr-o dată, chiar dacă textul programului așa sugerează. Iar dacă funcția **hPutStr** urmează să trimita la ieșire stringul (șirul de caractere) în blocuri de mărimi fixe, doar un bloc din fișierul de intrare trebuie să se afle în memorie la un moment dat (nu degeaba se numește „lazy evaluation” - evaluare leneșă). Fișierul de intrare este închis automat abia când a fost citit ultimul caracter.

7.5 Haskell și Programarea Imperativa

Ca ultimă observatie, programarea I/O ridică o problemă importantă: acest stil arată uimitor de asemănător cu programarea imperativă obișnuită. De exemplu, funcția **getLine**:

```
getLine =   do c <- getChar
               if c == '\n'
               then return ""
               else do l <- getLine
                       return (c:l)
```

este de o asemănare izbitoare cu acest cod scris imperativ (într-un limbaj imperativ, nu neapărat real):

```
function getLine() {
  c := getChar();
  if c == `\n` then return ""
            else {l := getLine();
                  return c:l}}
```

Deci, în final, a reinventat Haskell pur și simplu roata programării imperative?

În unele sensuri, da. Monada I/O constituie un mic sublimbaj imperativ în interiorul Haskell-ului și, astfel, componenta I/O a unui program poate părea asemănătoare cu obișnuitul cod scris imperativ. Dar există o diferență importantă: nu există semantici speciale,suplimentare, cu care utilizatorul să să aibă de-a face. În particular, semantica denotațională a Haskell-ului nu este compromisă. Aspectul imperativ al codului monadic într-un program nu se abate de la aspectul funcțional al limbajului Haskell. Un programator expert în

limbaje funcționale ar trebui să poată minimiza componenta imperativă a programului, utilizând ocazional monada I/O pentru o cantitate minimă de secvenţiere de nivel superior. Monada separă clar componenta funcțională a programului de cea imperativă. În contrast, limbajele imperative cu subseturi de instructiuni funcționale nu au, în general, nici o barieră bine definită între „lumile” pur-funcțională și imperativă (n.tr.: permit să amesteci cod imperativ în funcții, ceea ce le poate face să nu mai fie funcții în sens matematic).

8. Clase Standard Haskell

În această secțiune introducem clasele standard de tipuri predefinite în Haskell. Am simplificat oarecum aceste clase omițând unele dintre metodele de interes mai mic din aceste clase; raportul referitor la Haskell conține o descriere mai completă. De asemenea, unele dintre clasele standard sunt conținute de bibliotecile standard Haskell; acestea sunt descrise în cartea Haskell Library Report, accesibilă on-line.

8.1 Clasa tipurilor cu egalitate și Clasa tipurilor ordonate

Clasele **Eq** și **Ord** au fost deja discutate. Definiția clasei **Ord** în biblioteca **Prelude** este ceva mai complexă decât versiunea simplificată a clasei **Ord** prezentată anterior. În particular, notați metoda **compare**:

```
data Ordering =  EQ | LT | GT
compare       :: Ord a => a -> a -> Ordering
```

Metoda **compare** este suficientă pentru a defini toare celelalte metode (prin intermediul celor implicite) în această clasă și este cea mai buna metodă de a crea instanțe ale clasei **Ord**.

8.2 Clasa Tipurilor Enumerate

Clasa **Enum** are un set de operații care stau la baza notațiilor secvențelor aritmetice; de exemplu, notația secvenței aritmetice **[1,3..]** înlocuiește apelul de functie **enumFromThen 1 3** (vedeți capitolul 3.10 pentru interpretarea formală). Acum putem vedea că secvențele aritmetice pot fi folosite pentru a genera liste de elemente din orice tip care este instanță a clasei **Enum**. Aceasta include nu numai cele mai multe tipuri numerice, ci și tipul **Char**, astfel încât, de exemplu, **['a'..'z']** denotă lista literelor mici în ordine alfabetică. În plus, tipurile enumerate definite de utilizator precum **Color** pot fi introduse în clasa **Enum** printr-o declarație *instance*. Deci, în acest caz se va evalua:

[Red .. Violet] → [Red, Green, Blue, Indigo, Violet]

Notați că o astfel de secvență este *progresie aritmetică* în sensul că incrementul dintre valori este constant, chiar dacă valorile nu sunt numere. Multe tipuri din clasa **Enum** pot fi proiectate bijectiv pe submulțimi de numere întregi. De exemplu **fromEnum** și **toEnum** convertesc între **Int** și un tip din clasa de tipuri **Enum**. (n.tr.: Tocmai aceasta caracterizează tipurile enumerabile, sunt tipuri ale căror elemente pot fi numărate atribuindu-le indici întregi din **Z** sau **N**.)

8.3 Clasele Read și Show

Instanţele clasei ***Show*** sunt acele tipuri care pot fi convertite în şiruri de caractere (de obicei pentru operaţii I/O). Clasa ***Read*** oferă funcţii pentru citirea şirurilor de caractere cu scopul de a obţine valori pe care acestea le pot reprezenta. (Ex: stringul ''375'' poate reprezenta 2 întregi şi un număr real.) Cea mai comună funcţie din clasa ***Show*** este *show*:

```
show :: (Show a) => a -> String
```

În mod natural, *show* ia orice valoare a unui tip şi returnează reprezentarea acestuia ca şir de caractere (listă de caractere), ca în exemplul: *show (2+2)* a carui rezultat este "*4*". Acest lucru este în regulă, dar de obicei avem nevoie să creem şiruri mai complexe care pot conţine reprezentarea mai multor valori, ca în exemplul următor:

```
"Suma dintre " ++ show x ++ " si " ++ show y ++ " este " ++ show (x+y) ++ "."
```

dar sa ştiţi că la executie, o asemenea concatenare lungă ajunge să fie un pic ineficientă. De exemplu, putem considera o funcţie pentru a reprezenta arborele binar din Secţiunea 2.2.1 ca un şir, folosind marcaje adecvate pentru a arăta imbricarea subarborilor şi separarea ramurilor din stânga de cele din dreapta (cu condiţia ca tipul elementului să fie reprezentabil ca şir):

```
showTree :: (Show a) => Tree a -> String

showTree (Leaf x) = show x

showTree (Branch l r) = "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Deoarece (++) realizează un număr de operaţii elementare, proporţional cu lungimea stringului de afişat făcând concatenarea caracter cu caracter (matematicienii spun că are complexitate în timp liniară în raport cu lungimea argumentului său stâng), rezultă că funcţia showTree de mai sus consumă la evaluare un timp proporţional cu pătratul mărimii arborelui. (Un arbore de 100 ori mai bogat ar putea fi afişat într-un timp de 100 000 de ori mai mare!)

Pentru a o face rapidă (de complexitate liniară – se zice), funcţia *shows* o scriem astfel:

```
shows :: (Show a) => a -> String -> String
```

shows primeşte o valoare printabilă şi un şir, şi returnează acel şir cu reprezentarea valorii concatenată în faţă. Al doilea argument serveşte ca un fel de acumulator de şiruri , iar *show* poate fi acum definit ca un *shows* care porneste cu acumulatorul null. Aceasta este de altfel definiţia (implicită) a lui *show* în declaraţia clasei ***Show***:

```
show x = shows x ""
```

Putem folosi *shows* pentru a defini o versiune mai eficientă a lui *showTree*, care are de asemenea un argument acumulator pentru text (string).

```
showsTree :: (Show a) => Tree a -> String -> String
```

```
showsTree (Leaf x) s       = shows x s

showsTree (Branch st dr) s = '<' : showsTree st ('|' : showsTree dr ('>' : s))
```

Aceasta rezolvă problema eficienţei (*showsTree* are complexitate liniară), dar prezentarea, lizibilitatea acestei funcţii (şi a altora ca aceasta) poate fi îmbunătăţită. Iată cum: Mai întâi vom crea un tip sinonim:

```
type ShowS = String -> String
```

Acesta este tipul unei funcţii care returnează reprezentarea ca şir a ceva urmat de un şir acumulator. În al doilea rând, putem evita folosirea acumulatorilor şi aglomerarea parantezelor la capătul din dreapta a construcţiilor lungi, prin folosirea compunerii funcţionale (produsul de functii, operatorul „.” este o compunere):

```
showsTree :: (Show a) => Tree a -> ShowS

showsTree (Leaf x) = shows x

showsTree (Branch l r) = ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

Ceva mult mai important decât stilizarea codului este această transformare: am dezvoltat reprezentarea de la un nivel obiectual (în acest caz şiruri) la un nivel funcţional. Ne putem gândi la tipul ei ca şi cum am spune că *showsTree* transformă un arbore într-o funcţie de afişare.

Funcţii ca ('<' :) sau ("un şir" ++) sunt funcţii de afişare primitive, iar noi vom construi funcţii mai complexe folosind compunerea funcţiilor.

Acum că putem transforma arborii în şiruri de caractere, să ne putem problema reciprocă. Ideea de bază este folosirea unui parser de un tip *a*, care este o funcţie ce primeşte un şir şi returnează o listă de perechi *(a, String)*[9]. Biblioteca *Prelude* oferă un tip sinonim pentru astfel de funcţii:

```
type ReadS a = String -> [(a,String)]
```

Cand citirea decurge normal un parser returnează o listă de perechi formată dintr-o unică pereche conţinând o valoare a tipului *a*, citită din şirul de intrare şi (şi eventual structurată) pusă în pereche cu restul şirului ce urmează părţii parsate/analizate/citite. Daca parsarea nu a fost posibilă rezultatul este o listă goală [], iar dacă există cel puţin o parsare posibilă (o ambiguitate) lista rezultată conţine mai mult de o pereche. Funcţia standard *reads* este un parser pentru orice instanţă a lui *Read*:

```
reads :: (Read a) => ReadS a
```

Putem folosi această funcţie şablon pentru a defini o funcţie de parsare a reprezentărilor ca şir a arborilor binari produşi de *showsTree*. Listele reprezentate descriptiv - similare mulţimilor din matematică - ne permit să exprimăm convenabil modul de construire al unor astfel de parsere:[14]

```
readsTree :: (Read a) => ReadS (Tree a)

readsTree ('<':s) = [(Branch st dr, u) | (st, '|':t) <- readsTree s,
```

```
                                    (dr, '>':u) <- readsTree t ]

readsTree s = [(Leaf x, t) | (x,t) <- reads s]
```

Să examinăm un moment, în detaliu, definiţia acestei funcţii. Există două cazuri pe care le putem lua în considerare: dacă primul caracter al şirului de parsat este '<', ar trebui să avem reprezentarea unei ramuri; altfel, avem o frunză. În primul caz, apelarea restului şirului de intrare următor parantezei unghiulare deschise s, orice parsare posibilă trebuie să fie un arbore Branch st dr cu şirul rămas u, supus următoarelor condiţii:

1. Arborele st poate fi parsat de la începutul şirului s.
2. Şirul ce rămâne (urmând reprezentarea lui l) începe cu '|'. Apelarea cozii şirului curent t.
3. Arborele dr poate fi parsat de la începutul lui t.
4. Şirul ce rămâne în urma parsării începe cu '>', iar u este coada.

Observaţi puterea semnificativă obţinută din combinarea şablonului potrivit cu descrierea listei: modelul parserului rezultat este dat de expresia principală a descrierii listei, primele două condiţii anterioare sunt exprimate de primul generator ("(st, '|':t) este produs din lista de parsări a lui s"), iar condiţiile rămase sunt exprimate de al doilea generator.

A doua ecuaţie din definiţie doar precizează că pentru a parsa reprezentarea unei frunze, parsăm o reprezentare a tipului elementului arborelui şi aplicăm valorilor astfel obţinute constructorul *Leaf*.

---

[14] O abordare şi mai elegantă a parsării foloseşte monade şi parserele modulare. Acestea sunt părţi ale unei biblioteci standard de parsere, (n.tr. Cum este Parsec) distribuită împreună cu majoritatea sistemelor Haskell