3.3. Tipuri recursive și arbori polimorfi

Acest subcapitol discută un exemplu preluat din volumul în limba engleză “A Gentle Introduction to Haskell 98” scris de Paul Hudak, John Peterson și Joseph H. Fasel. Pentru cititorii români necunoscători ai limbii engleze (dar oare există informaticieni care să nu știe limba engleză ?) precizăm că Tree înseamnă Arbore, Leaf se traduce prin Frunză iar Branch prin Ramură deși în contextul din exemplu l-aș traduce mai curând prin Ramificație. Autori volumul amintit declară tipul Arbore astfel:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Ceea ce definește un arbore de ”a”-uri (cu etichete din tipul a) ca fiind sau o combinație dintre constructorul Leaf și un element de tip a sau o combinație dintre constructorul Branch și doi (sub)arbori de tipul (Tree a). Observați că după constructorul de date se poate scrie unul (sau mai multe) tipuri variabile (polimorfice). *Noutatea este că acestea pot fi nu numai variabile de tip dar și tipuri compuse polimorfice deci construcții sintactice care conțin constructori de tip*. Pe românește exemplul de mai sus s-ar scrie:

```
data Arbore a =Frunza a |
              Ramificatie  (Arbore a) (Arbore a)
```

O variantă puțin mai complexă a acestei declarații are avantajul de a determina interpretorul să afișeze și valorile din tipul nou creat. Cum faceți ? Adăugați la sfârșit: “deriving Show” !

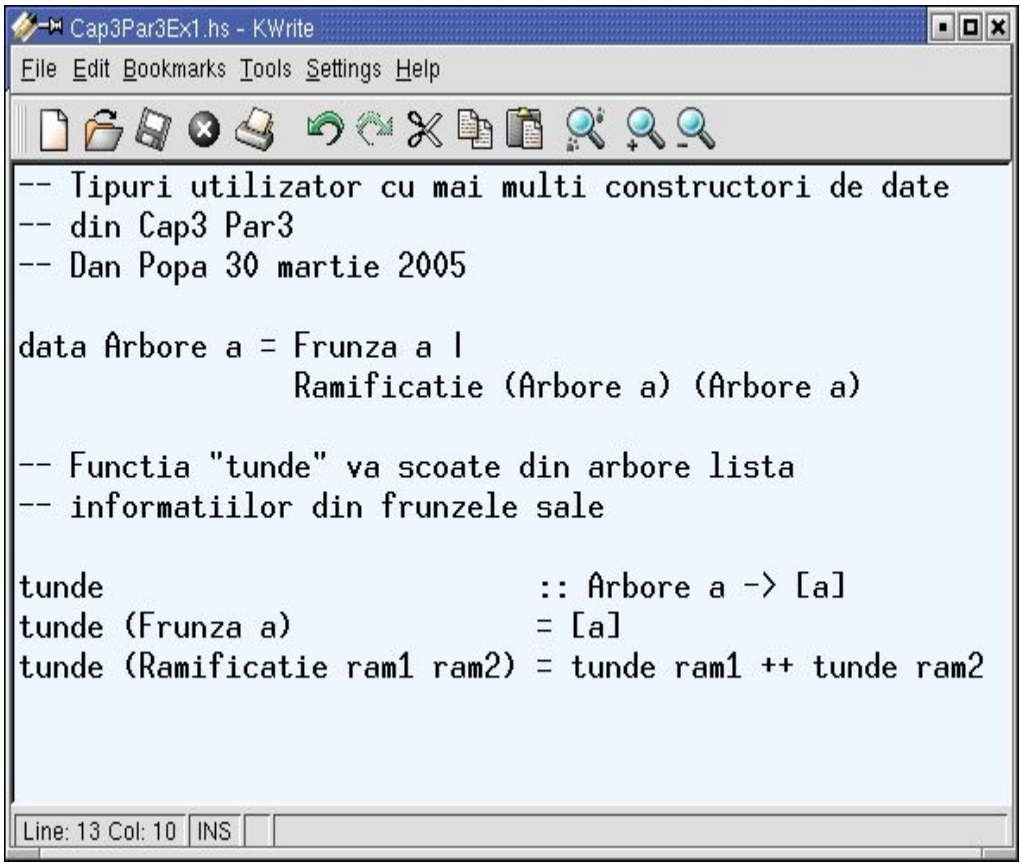
```
data Arbore a=Frunza a
             | Ramificatie (Arbore a) (Arbore a)
              deriving Show
```

Constructorii de date definiți în modul de mai sus acționează astfel:

Ramificație :: Arbore a -> Arbore a -> Arbore A

Frunză :: a -> Arbore a

Să prelucrăm un arbore ! Scrieți programul următor:



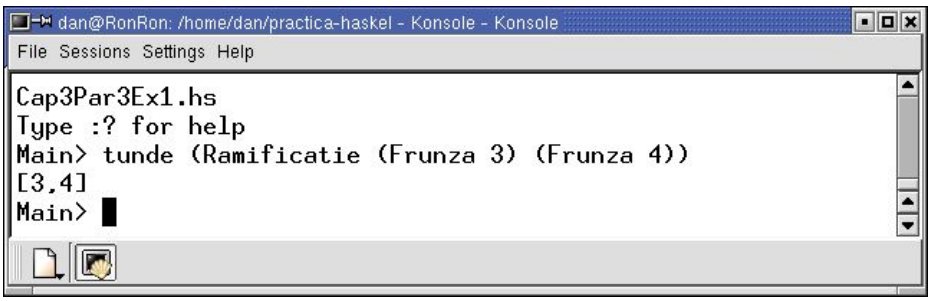
Câteva explicații sunt necesare:

1) Cuvântul Arbore este constructorul de tip și apare acolo unde

sintaxa permite să apară un tip. Un anume arbore concret este o dată, deci va fi construit cu constructorul de date. Un arbore cum este cel de mai jos, cu etichete întregi în frunze, deci de tipul Arbore Integer se va scrie așa:

(Ramificație (Frunza 3) (Frunza 4)

Și oferindu-l ca argument funcției ”tunde” veți primi ca răspuns lista frunzelor din arbore:



2) [a] este notație atât pentru tipul compus “listă de elemente de un tip oarecare a” cât și pentru constructorul de date [ ]. În acest din urmă caz [a] înseamnă “lista cu elementul al cărui valoare e dată de variabila a”. Valoarea acestui a este stabilită de pattern matching. Deci rândurl următor din program trebuie înțeles așa cum e explicat mai departe.

```
tunde :: Arbore a -> [a]
```

Funcția ”tunde” primește un argument de tipul “Arbore cu etichete de un tip a-oarecare” și returnează o listă de elemente de acel tip a-oarecare, același tip de elemente ca al etichetelor din frunzele arborelui dat. Toate frunzele arborelui au ca etichete valori de același

tip “a” și bineînțeles toate elementele listei rezultate sunt tot de tip “a”.
Haskell este, ați observat deja, un limbaj tipizat, cu toate consecințele care decurg de aici. Unei expresii i se evaluează nu numai valoarea dar și tipul. Ca urmare a procesului de aflare a tipului, fiecare expresie capătă un tip principal, ce poate fi întotdeauna precis inferat. Esența sistemului de tipuri în Haskell este dată de “sistemul de tipuri Hindley-Milner”. Această teorie a servit și servește ca bază formală pentru tipizarea din mai multe limbaje, inclusiv Haskell , ML, Miranda (ultimul este o marcă înregistrată a firmei Research Software Ltd.). Acest tip principal este *corect* în sensul că nu este nici excesiv de general dar nici nu omite vreun tip posibil, particular al acelei expresii. Sistemul de tipuri inferează automat acest tip principal. Pentru utilizator, acest tip este cel care cuprinde toate instanțele expresiei și nimic în plus, deci cel mai mic cu această proprietate.
Trecem la rândul următor:

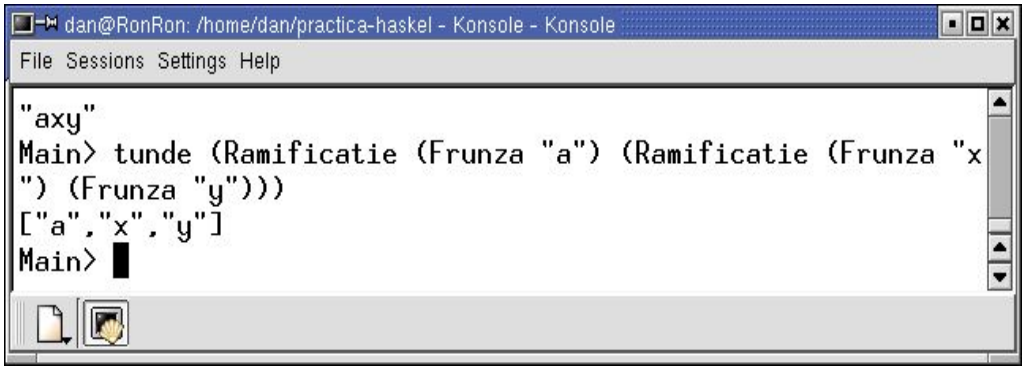
```
tunde (Frunza a)  = [a]
```

El trebuie înțeles astfel: “dacă argumentul (despre care știm că este de tipul Arbore a) este construit cu constructorul de date Frunza și o valoare a, atunci rezultatul este lista formată din acel element (de tipul a) care era plasat în nodul frunză.
Operatorul ++ este operatorul de concatenare a două liste (și a două stringuri). Adăugarea unui element într-o listă se face cu alt operator (:) care va fi prezentat în subcapitolul următor. Trecem la al doilea șablon de pattern-matching.

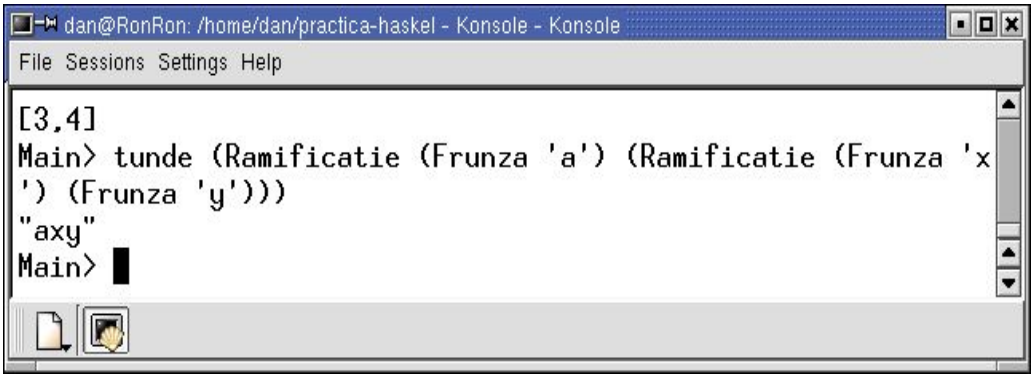
```
tunde (Ramificație ram1 ram2) = tunde ram1 ++ tunde ram2
```

Ea semnifică faptul că pentru un arbore cu subarbori legați la o ramificație lista frunzelor ce rezultă când se tunde arborele este concatenarea listelor frunzelor tunse din cei doi subarbori numiți ram1 și ram2. Funcția fiind recursivă, ieșirea din recursie se face în situația din cazul anterior.

Exemplu:



Surprinzător ne va părea exemplul:



În acest caz ne-am fi așteptat să obținem o listă de caractere, nu un string. Dar în Haskell listele de caractere și stringurile sunt același lucru ! E deci momentul să punem ordine în cunoștințele pe care le avem despre liste, ceea ce vom face în subcapitolul următor.