Chapter 1

Funcții recursive

Programarea funcțională este o paradigmă în care *funcțiile* ocupă un rol central, iar limbajele de programare care suportă această paradigmă au trăsături care permit, de exemplu, trimiterea unei funcții care parametru al unei altei funcții, crearea unei funcții la momentul execuției, compunerea de funcții ș.a.m.d. În acest sens, funcțiile sunt obiecte (în sens larg, nu în sensul programării orientate obiect) de prim rang într-un astfel de limbaj.

Deși se poate programa funcțional și în limbajele de nivel scăzut pe care le cunoașteți (de exemplu, C, C++, Java), limbajul Haskell conține un set de instrumente care fac programarea funcțională mai facilă.

1.1 Transparența referențială și efecte secundare

În cele mai multe limbaje de programare, *calculul* se face printr-o serie de *instrucțiuni*: e.g., adaugă 3 la variabila `x`, transmite un mesaj prin rețea, scrie un fișier ș.a.m.d. Aceste modificări ale stării sistemului se numesc *efecte*.

Spre exemplu, în limbajul de programare C, expresia `x = 3 + 4` se evaluează la 7, dar în plus, ca efect secundar, valoarea stocată în variabila `x` devine 7.

Tot ca exemplu, în limbajul C, expresia `printf(‘‘Hello’’)` se evalueaza la valoarea 5, dar în momentul evaluării, ca efect secundar, caracterele “Hello” sunt tipărite pe ecran.

Programarea funcțională încurajează scrierea de programe care nu au efecte secundare. În limbajele *pur* funcționale, cum este Haskell, nicio expresie nu poate avea un efect secundar. Orice calcul în Haskell se face prin evaluarea unor expresii, care prin definiție nu pot avea efecte secundare.

Acest lucru înseamnă inclusiv că în Haskell nu există variabile globale (nu aș putea să le schimb valoarea) și că nu există funcții care să *afișeze* ceva pe ecran (ar avea un efect secundar). Cu toate acestea, putem scrie în Haskell în principiu orice program pe care îl putem scrie într-un alt limbaj de uz general Turing-complet. Pentru moment, ne concentrăm pe programe care calculează un rezultat pornind de la un set de date de intrare. Mai târziu, vom vedea cum se împacă puritatea cu nevoia de a scrie programe care interacționează cu mediul (tastură, ecran, rețea ș.a.m.d.)

Avantajul unui limbaj funcțional pur este că acesta prezintă o proprietate interesantă și utilă numită *transparență referențială*. Transparența referențială înseamnă că o bucată de cod conduce la același rezultat indiferent de contextul în care apare. Acest lucru ușurează raționamentul despre programe și facilitează scrierea de cod lipsit de defecte (deși evident că oricine se străduiește suficient va reuși să scrie în Haskell și un bug).

1.2 Definirea funcțiilor în Haskell

Deoarece nu avem trăsături imperative (variabile globale, atribuire, bucle), singura metodă de calcul în Haskell este prin intermediul funcțiilor.

Funcțiile se definesc prin intermediul unor *ecuații*. Spre exemplu, funcția care calculează termenul din șirul lui Fibonacci de pe o poziție dată este definită prin următoarele trei ecuații:

```
fibo :: Integer -> Integer
fibo 0 = 1
fibo 1 = 1
fibo n = fibo (n - 1) + fibo (n - 2)
```

Pentru a efectua un calcul, ecuațiile care definesc o funcție sunt aplicate de sistem de la stânga la dreapta. Părțile stângi ale ecuațiilor se numesc *șabloane* (engl. patterns), iar primul șablon care se *potrivește* este folosit pentru efectuarea calculului.

Spre exemplu, așa se poate evalua expresia `fibo 3`:

1. `fibo 3 =`

   (Ecuația 3)

2. `fibo 2 + fibo 1 =`

   (Ecuația 3)

3. `(fibo 1 + fibo 0) + fibo 1 =`

   (Ecuația 2)

4. `(1 + fibo 0) + fibo 1 =`

   (Ecuația 1)

5. `(1 + 1) + fibo 1 =`

   (Ecuația 2)

6. `(1 + 1) + 1 =`

   (matematică)

7. `3`.

Acest gen de calcul se numește *raționament ecuațional* și poate fi folosit atât pentru a calcula o expresie (cum am făcut mai sus) cât și pentru a demonstra corectitudinea unui program referitor la o anumită specificație (vezi mai jos).

Sunt și alte moduri de a scrie funcția `fibo` într-un mod perfect echivalent:

1. folosind *gărzi* (ecuații condiționale):

   ```
   fibo :: Integer -> Integer
   fibo x | x == 0 || x == 1 = 1
   fibo n                    = fibo (n - 1) + fibo (n - 2)
   ```

2. folosind *gărzi*:

```
fibo :: Integer -> Integer
fibo n | n == 0    = 1
fibo n | n == 1    = 1
fibo n | otherwise = fibo (n - 1) + fibo (n - 2)
```

N.B. Ce este `otherwise`?

3. folosind *gărzi* (atenție la spațiul alb, care are o semantică bine definită):

```
fibo :: Integer -> Integer
fibo n | n == 0    = 1
       | n == 1    = 1
       | otherwise = fibo (n - 1) + fibo (n - 2)
```

4. folosind *if-then-else*:

```
fibo :: Integer -> Integer
fibo x = if x == 0 then 1
         else if x == 1 then 1
         else fibo (n - 1) + fibo (n - 2)
```

Ultima metodă nu este încurajată deoarece nu este un mod idiomatic de a defini funcții în Haskell. De asemenea, de cele mai multe ori preferăm folosirea șabloanelor în locul gărzilor.

Exerciții.

1. funcția factorial;

2. funcția cmmdc (prin scăderi repetate);

```
-- doua numere naturale ca argument, nu ambele 0
cmmdc :: Integer -> Integer -> Integer
cmmdc x 0 = x
cmmdc 0 y = y
cmmdc x y = if x > y then cmmdc (x - y) y else cmmdc x (y - x)
```

3. funcția cmmdc (prin scăderi repetate, folosind gărzi);

4. funcția cmmdc (prin scăderi repetate, folosind gărzi, fără if-then-else, cât mai puține ecuații, cât mai puține gărzi);

5. funcția cmmdc (prin împărțiri repetate);

6. funcția cmmmc (folosiți cmmdc).

1.3 Funcții cu acumulator

De multe ori, funcțiile recursive pe care le scriem direct pentru un anumit calcul sunt ineficiente.

```
sumn : Integer -> Integer
sumn 0 = 0
sumn n = n + sumn (n - 1)
```

Acest lucru este cauzat de folosirea stivei

1. `sumn 3 =`
2. `3 + sumn 2 =`
3. `3 + (2 + sumn 1) =`
4. `3 + (2 + (1 + 0)) =`
5. `3 + (2 + 1) =`
6. `3 + 3 =`
7. `6`.

O transformare cvasi-mecanică, adăugarea unui *acumulator*, este o optimizare care permite compilatorului să renunțe la folosirea unei stive, deoarece toate apelurile recursive sunt în poziția de coadă (engl. tail-position). Astfel de funcții sunt la fel de eficiente ca folosirea unei bucle while într-un limbaj de programare obișnuit.

```
sumna : Integer -> Integer -> Integer
sumna 0 a = a
sumna n a = sumna (n - 1) (a + n)
```

Din fericire, cele două funcții sunt echivalente funcțional (dacă alegem inițial `a = 0`), lucru care poate fi arătat folosind raționament ecuațional pentru a demonstra prin inducție faptul că `sumna n a = a + sumn n` pentru orice `n` natural:

1. `sumna n a =`
   (Ecuația 2 `sumna`)
2. `sumna (n - 1) (a + n) =`
   (Ipoteză inducție)
3. `(a + n) + sumn (n - 1) =`
   (Asociativitatea adunării)
4. `a + (n + sumn (n - 1)) =`
   (Ecuația 2 `sumn`)
5. `a + sumn n`.

N.B. Mai rămâne de arătat cazul de bază (`n` = 0).
N.B. Observați folosirea proprietății de transparență referențială.

1.4 Tuple

```
ghci> fst (8, 11)
ghci> snd (8, 11)
ghci> snd (8, True)
ghci> snd (8, (True, 11))
ghci> fst ('A', (True, 11))
ghci> fst ('A', (True, 11))
```

```
sum :: (Integer, Integer) -> Integer
sum (x, y) = x + y
```

1.5 Liste

Listele sunt un tip de date care permite agregarea mai multor valori de același tip.

```
ghci> length []
length []
0
ghci> length [1, 2, 5]
length [1, 2, 5]
3
ghci> length [1, 2, 5, 10]
length [1, 2, 5, 10]
4
ghci> head [1, 5, 19]
head [1, 5, 19]
1
ghci> tail [1, 5, 19]
tail [1, 5, 19]
[5,19]
ghci> [ [], [1], [2, 3, 4] ]
[ [], [1], [2, 3, 4] ]
[[],[1],[2,3,4]]
ghci> length [ [], [1], [2, 3, 4] ]
length [ [], [1], [2, 3, 4] ]
3
ghci> head [ [], [1], [2, 3, 4] ]
head [ [], [1], [2, 3, 4] ]
[]
ghci> head []
head []
*** Exception: Prelude.head: empty list
ghci> null []
null []
True
ghci> null [1, 2, 3]
null [1, 2, 3]
False
```

Funcțiile recursive care procesează liste se scriu folosind două *pattern*uri:

1. *pattern*ul `[]`, care se potrivește cu lista vidă;
2. *pattern*ul `(x:xs)`, care se potrivește cu orice listă nevidă: `x` ia valoarea primul element din listă (capul listei), iar `xs` coada listei (toată lista, mai puțin prim element).

Iată o funcție care calculează suma tuturor elementelor dintr-o listă:

```
sumlist'' :: [Integer] -> Integer
sumlist'' []     = 0                  -- patternul "[]" se potriveste peste lista vida
sumlist'' (x:xs) = x + sumlist'' xs   -- patternul "(x:xs)" se potriveste peste lista ca
                                      -- are primul element x si coada xs
```

Funcția de mai sus se poate scrie și folosind `head`, `tail`, `null`:

```
sumlist :: [Integer] -> Integer
sumlist l = if null l then -- de evitat
              0
            else
              (head l) + sumlist (tail l)

sumlist' :: [Integer] -> Integer
sumlist' l | null l    = 0 -- de evitat
           | otherwise = (head l) + sumlist (tail l)
```

Este preferată forma `sumlist''` (care folosește pattern matching) în dauna `sumlist`, `sumlist'` deoarece folosirea funcțiilor parțial definite `head`, `tail` poate predispune la erori.