Erlang feladatok

Patai Gergely*

2010. november 10.

Tartalomjegyzék

1 Lista hossza **1**

2 Számlista minden elemének növelése **2**

3 Egy lista utolsó elemének meghatározása **3**

4 Lista összes nemüres részlistáját tartalmazó lista **4**

1 Lista hossza

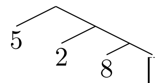
Egy lista elemeinek számát a lista hosszának nevezzük. Írjunk olyan eljárást, amely megfelel az alábbi fejkommentnek!

```
% @spec list_length(L::[any()]) -> H::integer().
% Az L lista hossza H.
```

A feladat megoldásához tisztában kell lenni a lista fogalmával. Az Erlangban, ahogy a Prologban is, a lista rekurzív adatszerkezet, amelyet a következő módon definiálhatunk:

```
% @type [any()] = [] | [any() | [any()]]
```

A definíció szerint egy lista vagy üres – ekkor a [] atommal jelöljük –, vagy egy [X | Xs] alakú struktúra, ahol X a lista első eleme, Xs pedig a további elemeket tartalmazó lista. Az X-et a lista *fejének*, az Xs-t pedig a lista *farkának* nevezzük. Például az $[5, 2, 8]$ lista reprezentációja a következő fastruktúra:



Ugyanez a lista felírható úgy is, hogy expliciten jelöljük a rekurzív struktúrát: [5 | [2 | [8 | []]]]. Erre általában akkor van szükség, amikor listát feldolgozó függvényt írunk, hiszen mintaillesztéssel tudjuk eldönteni, hogy az üres vagy a nemüres esettel van-e éppen dolgunk.

A rekurzív adatszerkezeteken értelmezett függvények általában szintén rekurzívan definiálhatók. Ehhez a következő általános gondolati sémát követhetjük:

- triviális esetek lefedése, amelyek nem igényelnek rekurziót;
- a feladat felbontása rögtön elvégezhető műveletekre, továbbá egy ugyanolyan alakú, de kisebb részfeladatra, amelynek az eredményéből a teljes eredmény egy lépésben előállítható.

Az is jellemző, hogy a feladat szempontjából érdekes esetek pontosan egybeesnek a rekurzív adatszerkezet definíciójában megadott alternatívákkal. Például listafüggvények esetén sokszor elég, ha a függvényt két klózzal definiáljuk: az üres és a nemüres esettel.

*Lektorálta, kiegészítette: Hanák Péter

A lista hosszát előállító függvény is ebbe a kategóriába tartozik. Első körben foglalkozzunk a triviális esettel: az üres lista elemeinek száma 0, tehát definíció szerint a *list_length* függvénynek is ezt kell visszaadnia, ha üres listára alkalmazzuk:

$list_length\ ([\,]) \rightarrow 0.$

A rekurzív eset leírásakor első lépésben tegyünk úgy gondolatban, mintha már működne a függvényünk! Ha a függvény bemenete és kimenete közötti kapcsolat jól definiált, akkor papíron „futtathatjuk" a csökkentett méretű feladatra, és utána elgondolkozhatunk azon, hogy ez az eredmény hogyan járul hozzá a teljes feladattól várt eredményhez.

Az biztos, hogy az összes nemüres esetet lefedhetjük a következő mintával:

$list_length\ ([X \mid Xs]) \rightarrow \ldots$

Mi kerüljön a jobb oldalra? Általában a jelenleginél összetettebb a feladat, és nehéz rögtön átlátni a teendőket. Ilyenkor sokat segít, ha egy konkrét példán keresztül dolgozzuk ki a megoldást.

Vehetjük például a fenti $[5, 2, 8]$ listát. Ha erre illesztjük a második esetben megadott mintát, X értéke 5 lesz, Xs-é pedig $[2, 8]$. A *list_length* függvény definíciója alapján az előbbire 3-at, az utóbbira 2-t várnánk eredményként. Mindezt fel is rajzolhatjuk:

$$\underbrace{[5,2,8]}_{3} \quad \overset{+1}{\Longleftarrow} \quad [5|\underbrace{[2,8]}_{2}]$$

A rekurzív kiértékelés eredményének felhasználásához a következő kérdésre kell válaszolnunk: milyen kapcsolatban van a teljes listára adandó 3 eredmény a rövidebbre kapott 2-vel? Ebben az esetben a válasz triviális: ha egy lista elé fűzünk egy tetszőleges elemet, akkor a hosszát eggyel növeltük, tehát a rekurzív hívás eredményéhez is 1-et kell adnunk:

$list_length\ ([_X \mid Xs]) \rightarrow list_length\ (Xs) + 1.$

Ezen a ponton gyakorlatilag kész is vagyunk. A két esetet egymás után leírva megkapjuk a végső megoldást. Csak arra kell figyelnünk, hogy a központozási jelek rendben legyenek, azaz az első esetet pontosvesszővel, a másodikat pedig ponttal zárjuk le:

```
list_length ([]) →
  0;
list_length ([_X | Xs]) →
  list_length (Xs) + 1.
```

Ebben az átírásban már követtük azt a konvenciót, hogy új sort kezdünk a klózok törzsének, még akkor is, ha csak egy kifejezésből állnak.

2 Számlista minden elemének növelése

A fenti mintát követve oldjuk meg a következő feladatot:

```
% @spec list_inc(L::[integer()]) -> IL::[integer()]
% Az IL egészlista az L egészlistának olyan másolata, amelynek
% ugyanannyi eleme van, mint L-nek, de az IL minden elemének értéke
% pontosan eggyel nagyobb, mint az L megfelelő elemének az értéke.
```

Példa:

$list_inc\ ([5, 2, 8]) == [6, 3, 9].$

A fenti megfontolások alapján az első kérdés az, hogy jól járunk-e, ha az üres és nemüres eseteket választjuk szét? A válasz „igen", hiszen az üres listára rögtön meg tudjuk mondani a végeredményt:

$list_inc\ ([\,]) \rightarrow [\,].$

A nemüres eset kicsit összetettebb, mint az előző feladatban volt, mert a rekurzív hívás eredménye mellett azt a részét is fel kell használni a listának, amelyet a rekurzív hívás nem kap meg. Megint csak vegyünk egy konkrét példát! Bal oldalt a kívánt végeredményt láthatjuk, jobb oldalt pedig a különböző részekből eredeztethető eredményeket.

$$\underbrace{[5,2,8]}_{\displaystyle\downarrow \atop \displaystyle [6,3,9]} \quad \xleftarrow{[\cdot|\cdot]} \quad [5|\underbrace{[2,8]}]\ \ \ \ \ \ \begin{matrix}\downarrow & \downarrow \\ 6 & [3,9]\end{matrix}$$

Látható, hogy a rekurzív hívás eredménye megegyezik a teljes eredmény farkával, az eredmény fejét pedig a bemenet fejéből kaphatjuk meg úgy, hogy eggyel növeljük az értékét. A két eredményt a fej-farok jelölést felhasználva kombinálhatjuk:

list_inc ([X | Xs]) → [X + 1 | *list_inc* (Xs)].

Utolsó lépésként az előző feladathoz hasonlóan egymás után leírhatjuk a fenti két klózt:

```
list_inc ([]) →
  [];
list_inc ([X | Xs]) →
  [X + 1 | list_inc (Xs)].
```

3 Egy lista utolsó elemének meghatározása

Vegyük a következő specifikációt:

```
% @spec last(L::[any()]) -> E::any().
% Az L lista utolsó eleme E.
```

Példa:

last ([5, 1, 2, 8, 7]) == 7.

Mi az eredmény, ha a bemenet az üres lista? Az előző két esettel ellentétben a válasz itt a legkevésbé sem nevezhető triviálisnak, hiszen egy üres listának definíció szerint *nincs* utolsó eleme. Más szóval a *last* függvény a [] bemenetre nem definiált, ahogy például a valós számokon értelmezett négyzetgyök sem definiált negatív számokra.

Egyelőre ne foglalkozzunk a nem definiált tartománnyal, hanem válaszoljunk arra a kérdésre, hogy ha nem az üres lista, akkor mi tekinthető triviális esetnek? Mi az a bemenet, amelyre egy lépésben meg tudjuk mondani az eredményt? A válasz az egyelemű lista:

last ([X]) → X.

Az egyelemű lista utolsó eleme természetesen maga az egyetlen elem. A rekurzív eset levezetéséhez vegyük a megadott példát:

$$\underbrace{[5,1,2,8,7]}_{7} \quad = \quad [5|\underbrace{[1,2,8,7]}_{7}]$$

Nyilvánvaló, hogy nem befolyásolja a lista utolsó elemét, ha az elejére beszúrunk egy plusz elemet, tehát a rekurzív hívás eredménye egyben a végeredmény is:

last ([_X | Xs]) → *last* (Xs).

És a teljes megoldás:

```
last ([X]) →
  X;
```

```
last ([_X | Xs]) ->
  last (Xs).
```

No de mi legyen az üres listával? A válasz a kontextustól függ: ha a *last* függvény használatakor az adott programban normális esetnek számít az üres bemenet, akkor érdemes definiálni egy „biztonságos” változatot, amely a nem definiált tartományokat is lefedi:

```
safe_last ([X]) ->
  {ok, X};
safe_last ([_X | Xs]) ->
  safe_last (Xs);
safe_last (_) ->
  error.
```

A helyes hívások eredményét párba csomagoljuk az *ok* atommal, míg a nem definiált esetet az *error* atommal jelezzük. A kétféle visszatérési értéket a szokásos módon, mintaillesztéssel különböztethetjük meg.

A másik lehetőség az, hogy maradunk az egyszerű *last*-nál. Ekkor a hibás bemenet kivételt okoz, amit valahol máshol kell lekezelnünk, például egy monitorozó processzben. Ez a kérdés azonban már kívül esik a funkcionális programozás témakörén.

4 Lista összes nemüres részlistáját tartalmazó lista

Vegyünk most egy kicsit összetettebb feladatot:

```
% sublists(Xs::[any()]) -> Pss::[{B::integer(), Ps::[any()], A::integer()}].
% A Pss lista elemei olyan {B,Ps,A} hármasok, amelyekben a Ps lista az Xs
% lista olyan folytonos, nemüres részlistája, amely előtt B és amely után
% A számú elem áll az Xs-ben.
```

Példa:

$$sublists([a,b,c]) == [\{0,[a],2\},\{1,[b],1\},\{2,[c],0\},\{0,[a,b],1\},\{1,[b,c],0\},\{0,[a,b,c],0\}].$$

Van-e értelme az üres listának? Ha a definíciót követjük, akkor igen, hiszen a következő állítást kell csak megfogalmaznunk a specifikáció által megszabott módon: az üres listának nincs nemüres részlistája. Más szóval – kicsit nyakatekertebben, de egyben erlangosabban is – az üres lista nemüres részlistáinak listája az üres lista.

```
sublists ([]) -> [].
```

Most lássuk, meg tudjuk-e oldani a feladatot úgy, hogy az összes többi listát egységesen kezeljük. Az eddigi mintát követve vegyük a példabemenetet, és vessük össze a rekurzív hívás eredményét a teljes végeredménnyel!

$$\begin{aligned} sublists([a,b,c]) &= [\{0,[a],2\},\{1,[b],1\},\{2,[c],0\},\{0,[a,b],1\},\{1,[b,c],0\},\{0,[a,b,c],0\}] \\ sublists([b,c]) &= [\{0,[b],1\},\{1,[c],0\},\{0,[b,c],0\}] \end{aligned}$$

Hogyan kaphatjuk meg a második listából és a még fel nem használt a elemből az első listát? Gondoljuk végig, mit történik a részlisták listájával, ha egy elemmel bővítjük a bemenő lista elejét! A teljes eredményben kétféle részlista fordulhat elő:

- olyan, amelyik már a rövidebb listának is részlistája volt; és
- olyan, amelyik tartalmazza az új elemet.

Az első csoport elemei egyértelműen megfeleltethetők a rekurzív hívásban kapott lista elemeinek, csupán a megelőző elemek számát kell növelni eggyel, hiszen a nagy listában ezek a részlisták eggyel jobbra tolódnak. A második csoport elemei pedig szükségszerűen a bemenő lista prefixumai (első elemétől kezdődő részlistái), hiszen csak egybefüggő részlistákat sorolunk fel. A két lista elemeit tetszőleges sorrendben kombinálva megkapjuk a végeredményt.

A fenti műveletsort felülről lefelé (top-down) haladva írhatjuk le, menet közben fejtve ki a részleteket. A végeredmény tehát a két csoport kombinációja:

$$\mathit{sublists}\ (\mathsf{L} = [_\mathsf{X} \mid \mathsf{Xs}]) \to \mathit{prefixes}\ (\mathsf{L}) \mathbin{++} \mathit{shift}\ (\mathit{sublists}\ (\mathsf{Xs})).$$

Az $\mathsf{L} = [_\mathsf{X} \mid \mathsf{Xs}]$ minta ún. *réteges minta*: az egyenlőségjel jobb oldalára illeszkedik, de a teljes argumentum elérhető a bal oldalon megadott néven is. Tehát a teljes átadott lista L, a feje X, a farka pedig Xs.

A *prefixes* függvény feladata, hogy előállítsa a lista prefixumait, és mindegyikhez mellékelje a hátralévő elemek számát, továbbá a megelőző elemekét is, ami persze definíció szerint 0.

Példa:

$$\mathit{prefixes}\ ([a, b, c]) == [\{0, [a], 2\}, \{0, [a, b], 1\}, \{0, [a, b, c], 0\}].$$

A *shift* függvény dolga csupán annyi, hogy a rekurzív hívás eredményében eggyel növelje a megelőző elemek számát.

Példa:

$$\mathit{shift}\ ([\{0, [b], 1\}, \{0, [b, c], 0\}, \{1, [c], 0\}]) == [\{1, [b], 1\}, \{1, [b, c], 0\}, \{2, [c], 0\}].$$

Ezeket a függvényeket a fentihez hasonló módon lehet definiálni. A kettő közül a *shift* az egyszerűbb; a szerkezete teljesen ugyanolyan, mint például a korábban látott *list_inc*-é, ezért itt nem is részletezzük.

A *prefixes* már valamivel bonyolultabb. Itt is az üres listával kezdünk: a definícióból világosan látszik, hogy ekkor az eredmény is üres:

$$\mathit{prefixes}\ ([\,]) \to [\,].$$

A rekurzív eset vizsgálatához vegyük a fenti példát. A korábban látottakhoz hasonlóan felírhatjuk a következő kapcsolatot, ahol a kettős nyíl bal oldalán az $[a, b, c]$ lista, a jobb oldalán pedig a $[b, c]$ lista prefixuma áll:

$$[\{0, [a], 2\}, \{0, \overbrace{[a, b}^{[a,b,c]}], 1\}, \{0, [a, b, c], 0\}] \quad \Longleftarrow \quad [a| \; [\{0, \overbrace{[b], 1\}, \{0, [b, c}^{[b,c]}], 0\}] \;]$$

Milyen transzformációt jelöl a kettős nyíl? Láthatóan egyszerű dologról van szó: a *prefixes* $([a, b, c])$ hívás eredménylistájának feje egy olyan hármas, amely a 0-ból, azaz a prefixumot megelőző elemek számából, az argumentumlista egyelemű listába csomagolt fejéből és az argumentumlista farkának hosszából áll. Az eredménylista farkát a rekurzív hívás eredményéből kapjuk, mégpedig úgy, hogy az utóbbiban a hármasokban lévő listák elé fűzzük a rekurzív hívásból kihagyott elemet, azaz az eredeti argumentumlista a fejét.

$$\mathit{prefixes}\ ([\mathsf{X} \mid \mathsf{Xs}]) \to [\{0, [\mathsf{X}], \mathit{length}\ (\mathsf{Xs})\} \mid \mathit{prepend}\ (\mathsf{X}, \mathit{prefixes}\ (\mathsf{Xs}))].$$

Az eredménylista farkának előállításához, azaz a kimaradt elem beszúrásához bevezettük a *prepend* műveletet. Lássunk egy példát az alkalmazására:

$$\mathit{prepend}\ (a, [\{0, [b], 1\}, \{0, [b, c], 0\}]) == [\{0, [a, b], 1\}, \{0, [a, b, c], 0\}].$$

Ez a függvény a *shift*-hez hasonlóan, a *list_inc* mintájára definiálható, így a gondolatmenetet nem írjuk le újra. Ehelyett álljon itt a *sublists* teljes definíciója az összes segédfüggvénnyel:

```
sublists ([]) ->
  [];
sublists (L = [_X | Xs]) ->
  prefixes (L) ++ shift (sublists (Xs)).

prefixes ([]) ->
  [];
prefixes ([X | Xs]) ->
  [{0, [X], length (Xs)} | prepend (X, prefixes (Xs))].

prepend (_, []) ->
  [];
```

```
prepend (X, [{B, Xs, A} | L]) →
    [{B, [X | Xs], A} | prepend (X, L)].


shift ([]) →
    [];
shift ([{B, X, A} | L]) →
    [{B + 1, X, A} | shift (L)].
```

Megjegyzés: mind a *shift*, mind a *prepend* kiváltható a *lists*:*map* függvénnyel vagy a listanézettel, így sokkal rövidebb kódot kapunk:

```
sublists ([]) →
    [];
sublists (L = [_X | Xs]) →
    prefixes (L) ++ [{B + 1, Y, A} || {B, Y, A} ← sublists (Xs)].


prefixes ([]) →
    [];
prefixes ([X | Xs]) →
    [{0, [X], length (Xs)} | [{B, [X | Ys], A} || {B, Ys, A} ← prefixes (Xs)]].
```