Laboratorul 3

Elm - Tipuri de date și funcții utile încorporate

Goluri

În acest laborator vei învăța să:

- 1. Utilizați variabile de tip pentru a defini tipurile și funcțiile generice
- 2. Utilizați constrângeri de tip pentru a restricționa tipurile pe care le poate lua o variabilă de tip
- 3. Utilizați tipul Poate pentru a exprima anulabilitatea (posibilitatea ca o valoare să lipsească)
- 4. Utilizați tipul de rezultat pentru a indica posibilitatea de eșec
- 5. Construiți, transformați și procesați liste

Resurse

Tabelul 3.1: Resurse de laborator

Resursă

Legătură

Prezentare generală a limbajului de bază Elm https://guide.elm-lang.org/core_language.html Biblioteca de be/ză Œ5t/h https://package.elm-lang.org/packages/elm/

3.1 Variabile de tip și constrângeri

Observați neconcordanțe în semnăturile funcției deduse?

```
Elm REPL

> isEq a b = a == b

<funcție> : a -> a -> Bool

> isGt a b = a > b

<funcție> : comparabil -> comparabil -> Bool
```

Unele nume de tip sunt litere mici (cum ar fi un), unele sunt nume cu minuscule (cum ar fi comparable) si unele sunt PascalCase (cum ar fi Bool). Literele mici sunt variabile de tip, iar numele mici sunt constrângeri de tip. Acestea ne ajută să scriem cod care este generic pe mai multe tipuri.

3.1.1 Variabile de tip

O variabilă de tip este o variabilă care variază peste tipuri. Poate fi folosit în semnăturile funcțiilor, așa cum ați văzut mai sus pentru isEq . Numele variabilelor de tip sunt litere mici și pot conține mai mult de un caracter (t1 tip1 sunt toate nume valide). (elem),

În semnăturile de funcție, variabilele de tip pot apărea de mai multe ori pentru a indica faptul că aceste valori trebuie să aibă același tip (dar acest tip poate fi de orice tip).

```
Lista 3.1.1 din Generic.elm (reverseTuple)

4 reverseTuple : (a, a) -> (a, a) 5 reverseTuple
(a, b) = (b, a)
```

Poate fi folosit și în declarațiile de tip pentru a crea containere generice. Cel mai simplu astfel de tip este doar un înveliș în jurul altui tip:

```
Lista 3.1.2 din Generic.elm (Cutie)

9 tip Box a = Box a
```

Pentru a folosi acest tip putem scrie următoarea funcție (în mare parte inutilă):

```
Listarea 3.1.3 din Generic.elm (unboxInt)

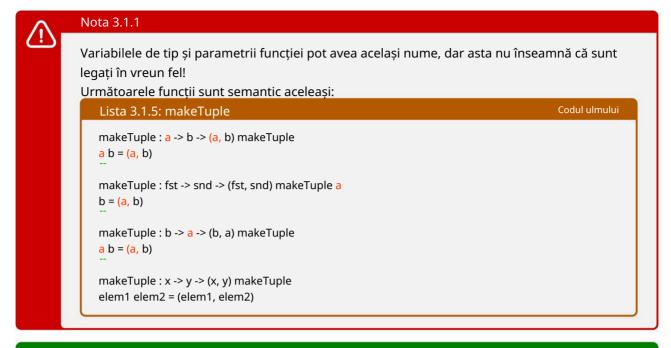
13 unboxInt : Box Int -> Int 14 unboxInt
(Caseta i) = i
```

Un model comun cu astfel de containere este utilizarea alias-urilor de tip pentru a crea tipuri pentru "configurații" comune:

```
Lista 3.1.4 din Generic.elm (pairAlias)

Codul ulmului

18 tip Pair a b = Pair a b 19 tip alias
IntPair = Pair Int Int 20 21 addIntPair : IntPair ->
Int 22 addIntPair (Pereche a b) = a + b
```



Întrebarea 3.1.1

Ce concept Java este echivalent cu variabilele de tip? Dar C++?

3.1.2 Constrângeri de tip

Există de fapt o restricție legată de numele variabilelor de tip: ele nu pot fi 1 : "apendice", "număr" sau "comparabile" deoarece acestea sunt numele de constrângeri de tip. O variabilă de tip în sine înseamnă "orice tip", care uneori poate fi prea general pentru a fi util.

Luați în considerare funcții precum compararea și operatorii de comparare, (um), (>), care ar trebui să fie ar fi (<) capabili să compare oricare două instanțe de același tip. Să le verificăm semnătura:

```
Elm REPL
> (<)
<funcție>: comparabil -> comparabil -> Bool > compara
<funcție>: comparabil -> comparabil -> Comandă
```

Aici comparabil indică faptul că: primul și al doilea parametru trebuie să aibă același tip și trebuie să fie comparabili.

Ca exemplu final, luați în considerare funcția par Add care ia două perechi de numere și adună cele două elemente ale fiecărui tuplu:

```
Elm REPL
> parAdd (a, b) (c, d) = (a + b, c + d) <funcție> :
( număr1, număr1 ) -> ( număr, număr ) -> ( număr1, număr )
```

Observați cum în tipul dedus elementele primului tuplu și rezultatul lor sunt toate legate, deoarece au tipul număr1 (cu 1), în timp ce elementele celui de-al doilea tuplu și rezultatul lor au un număr de tip (fără un 1).

¹De fapt, ele nu pot fi nume rezervate sau numele rezervate de mai sus cu un număr după ele (numărul 1).

Principalele constrângeri de tip arbore din Elm sunt:

appendable: tipuri care pot fi anexate: String şi List

number: tipuri care sunt numere: Int şi Float

comparabil: tipuri care pot fi comparate: numere, Char şi tuplu de , ir , Lista de comparabile, comparabile.



Nota 3.1.2

În Elm, nu există (prin proiectare) nicio modalitate de a face noi tipuri definite de utilizator care să satisfacă cele 3 constrângeri de tip încorporate. Puteți implementa funcții personalizate, dar acestea trebuie utilizate în mod explicit.

Întrebarea 3.1.2

**

Putem constrânge variabilele de tip în Java sau C++? Daca da cum?

3.2 Egalitatea

Conceptul 3.2.1: Tipuri de egalitate

Limbajele de programare implementează adesea două tipuri de egalitate:

Egalitatea referințelor: este comparată doar adresa către care indică referințele (pointerii).

Egalitatea structurală: se compară conținutul (câmpurile) celor două obiecte, folosind funcțiile de egalitate ale acestora, care, în funcție de limbaj, pot efectua diferite tipuri de comparații: Profund: sunt comparate doar câmpurile obiectului de nivel superior, folosind referință. egalitate Profunzime: fiecare câmp este comparat recursiv cu egalitatea profundă, folosind egalitatea sa

funcție (care, de exemplu, poate compara datele octet cu octet)

Aceste funcții de egalitate pot fi și personalizate (de exemplu, suprascrise în Java).

După ce ați văzut lista de mai sus, probabil vă întrebați: "nu există nicio constrângere de tip pentru egalitate?" sau "pentru a verifica dacă două valori sunt egale, trebuie să fie definită și o relație de ordine?".

În mod implicit, Elm implementează automat egalitatea structurală profundă pentru toate valorile prin operatorul = ... Aceasta înseamnă că putem compara întotdeauna două instanțe de același tip pentru egalitate.

Conceptul 3.2.2: Imuabilitate și egalitate

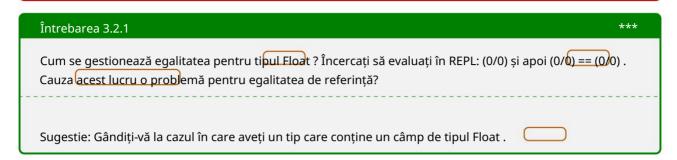
Când avem date imuabile, cu unele optimizări inteligente, putem întotdeauna să verificăm egalitatea structurală folosind egalitatea de referință.

Acest lucru se datorează faptului că știm că datele nu sunt niciodată modificate direct, dar este creată o nouă copie cu modificările. Dacă ne asigurăm că nu avem două copii (adică o serie de octeți reprezentând aceleași date, în două regiuni diferite de memorie) ale acelorași date, putem folosi întotdeauna egalitatea de referință pentru a verifica egalitatea structurală completă (profundă).



Nota 3.2.1

Implementarea implicită pentru == nu poate fi modificată. Pentru a compara doar un subset de câmpuri, trebuie să implementați propria funcție de egalitate personalizată și să o utilizați în mod explicit.

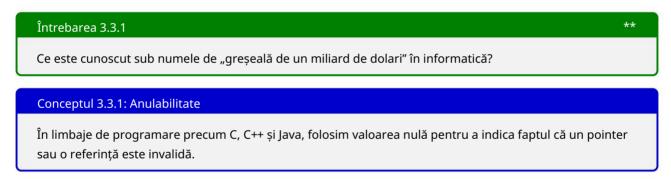


3.3 Studiu de caz: tipuri pentru tratarea erorilor

Acum că înțelegem variabilele de tip, să înțelegem în sfârșit câteva tipuri mai utile și mai interesante.

În acest moment, ar trebui să știți suficient pentru a naviga în documentația bibliotecii standard Elm aici: https://package.elm-lang.org/packages/elm/core/1.0.5/Basics.

3.3.1 Semnalizarea posibilității de anulare: tipul Maybe



În capitolul 1 am observat că funcțiile trebuie să gestioneze orice intrare posibilă. Dar ce se întâmplă atunci când nu există o ieșire bine definită pentru o anumită intrare?

Luați în considerare un faimos caz de împărțire cu 0: ce ar trebui să se întâmple când încercăm să împărțim la zero?

```
Elm REPL
> 1/0
Infinit : plutitor > 1 //
0
0 : int
> modBy 0 10
Error: Nu se poate efectua modul 0. Eroare de împărțire prin zero.
```

După cum puteți vedea, obținem rezultate diferite în funcție de tipurile (Int sau Float) cu care lucrăm, inclusiv chiar și o eroare de rulare, ceva ce nu ar trebui să se întâmple în Elm.

Un alt exemplu ar fi funcția stârc din cap<mark>itolul 2) Funcția nu este 100% corectă, pentru că nu numai că o putem numi cu argumente care reprezintă trei laturi care nu pot.</mark>

formează un triunghi (ex . stârc 1 1 10), dar îl putem numi și cu argumente negative! Deci, ce ar trebui să returneze funcția în acest caz?

Un prim gând ar putea fi să returnezi doar 0, semnalând că probabil ceva nu este în regulă. Dar acest lucru creează confuzie în cazul în care numim funcția cu argumente care reprezintă laturile unui triunghi degenerat, format din punct coliniar (adică stârcul 2 2 4 ar trebui să fie 0 pentru că aria este 0, nu pentru că intrarea este invalidă).

O altă soluție pentru a diferenția mai bine între ieșirile pentru intrare validă și nevalidă este să returnezi un rezultat care ar trebui să fie imposibil de obținut de la intrări valide, cum ar fi clasicul -1. Acest lucru rezolvă problema anterioară a ieșirilor ambigue, dar creează o nouă problemă: această convenție trebuie să fie documentată în mod clar și luată în considerare atunci când se utilizează funcția. Erori precum uitarea de a testa rezultatul și transmiterea acestuia unei funcții care se așteaptă la un număr întreg pozitiv se pot strecura cu ușurință.

Soluția în programarea funcțională este să folosiți tipuri enumerate pentru a reprezenta cele două rezultate posibile:

un rezultat bine definit

nici un rezultat

Acest concept este implementat în Elm cu tipul Maybe:

```
Lista 3.3.1: poate definiția tipului

tip poate a
= Doar a
| Nimic
```

Varianta Just reprezintă cazul rezultatului bine definit, iar varianta Nimic reprezintă cazul "fără rezultat".

Principalul avantaj al utilizării acestui tip este că trebuie să verificăm în mod explicit rezultatul funcției, altfel vom obține o eroare de compilare.

De exemplu, să rezolvăm problemele cu funcția heron schimbând ti<mark>pul de r</mark>eturnare la Maybe Float și returnând Nimic dacă argu<mark>mentele sunt negative sau nu pot forma un ţ</mark>riunghi valid:

```
Lista 3.3.2 din Shape.elm (validTriangle, safeHeron)

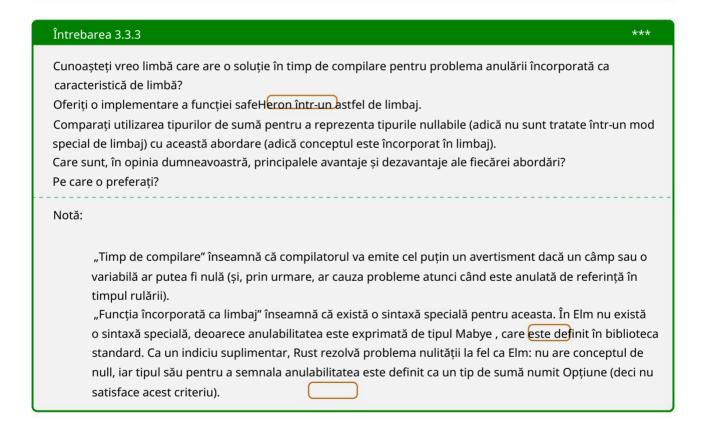
39 triunghi valid a bc = 40 ((a > 0)
&& (b > 0) &&cf(&&20)) && 44 ((a & & c) >= a)) 45
safeHer®afeHeron>aFbcat 47 (48249-50) Aaybe Float 46

dacă nu (validTriangle a bc) atunci
Nimic
altceva
Doar (heron a bc)
```



Acum, când încercăm să folosim funcția cu valori de intrare nevalide, returnează Nimic , ceea ce înseamnă că rezultatul este nedefinit. Când intrările sunt valide, rezultatul este returnat împachetat în varianta Just .

Întrebarea 3.3.2 ** Putem ști în momentul compilării dacă orice indicator este nul sau nu în C? Referențele C++ sunt diferite?



3.3.2 Semnalizarea posibilității de defecțiune: tipul Rezultat

Abordarea anterioară (adică folosirea Maybe) are un dezavantaj: aceeași variantă de returnare este folosită pentru a reprezenta toate erorile. Prin urmare, atunci când pot apărea mai multe erori, nu știm cu exactitate care dintre ele a cauzat eșecul funcției.

Pentru a rezolva această problemă, putem folo<mark>si tipul de</mark> rezultat , care poate include un rezultat de succes în Vajianta Ok și un tip de eroare în varianta Err :

```
Lista 3.3.3: Definiția tipului de rezultat

tip Valoare eroare rezultat

= Ok valoare

| Err eroare
```

Un caz de utilizare bun pentru acest tip ar fi funcți<mark>a de z</mark>onă, care poate eșua dacă:

raza cercului este negativă

oricare dintre lățimea sau înălțimea dreptunghiului este negativă

laturile triunghiului sunt negative sau nu pot forma un triunghi valid

Există două abordări principale pentru returnarea erorilor:

Utilizarea unui șir pentru a returna un mesaj de eroare

```
Lista 3.3.4 din Shape.elm (safeArea)
19 safeArea : Shape -> Result String Float
20 safeArea shape =
21 forma carcasei de
           Raza cercului ->
             dacă raza < 0 atunci
23
24
                Err "Raza cercului negativ"
25
             altfel
26
                 Ok (pi * rază * rază)
27
           Lățimea dreptunghiului înălțimea ->
28
              dacă (lățimea < 0) || (înălțime < 0) atunci
29
                Err "Lățimea sau înălțimea dreptunghiului negativ"
30
31
                 Ok (latime * inaltime)
32
           Triunghiul a bc ->
33
             case safeHeron a bc of
34
                 Doar zona -> Ok zona
35
                 Nimic -> Err "Laturile nu pot forma un triunghi"
```

Pentru fiecare caz de eroare fie returnăm rezultatul împachetat în varianta Ok, fie returnăm o eroare mesaj care precizează problema în varianta Er.

Vom verifica rezultatul safeHeron cu o expresie caz pentru a vedea dacă funcția a returnat a valoare în varianta Just , caz în care o reîncărcăm în varianta Ok din Result sau în Nimic variantă, când vom ști că funcția a eșuat pentru că laturile nu pot forma un valid triunghi.

```
> safeArea (Triunghi 2 2 3)
Ok 1.984313483298443 : Rezultat şir flotant
> safeArea (Triunghi 2 2 10)
Err ("Laturile nu pot forma un triunghi") : Rezultat String Float
> safeArea (Dreptunghi 2 10)
Ok 20: Rezultat String Float
> safeArea (Dreptunghi 2 -10)
Err ("Lățime sau înălțime negativă") : Rezultat șir flotant
> safeArea (Cercul 2)
Ok 12.566370614359172 : Rezultat șir flotant
> safeArea (Cercul -2)
Err ("Raza cercului negativ") : Rezultat șir flotant
```

Definirea unui tip de enumerare care reprezintă fiecare eroare posibilă

Abordarea anterioară are încă o problemă: apelantul funcției trebuie să proceseze șir returnat pentru a putea gestiona efectiv eroarea.

Pentru a rezolva această problemă, putem defini un tip enumerat cu o variantă pentru fiecare eroare, adică apelantul poate potrivi modelul pentru a gestiona eroarea în mod programatic.

De exemplu, pentru a reprezenta posibilele erori care pot apărea la calcularea ariei lui a triunghi, putem defini următoarele tipuri pentru a reprezenta toate erorile posibile:

```
Lista 3.3.5 din Shape.elm (InvalidTriangleError, TriangleSide)

67 tip InvalidTriangleError

68 = NegativeSide TriangleSide

69 | Triunghi imposibil

73 tip TriangleSide = A | B | C
```

Și apoi returnați tipul potrivit pentru fiecare eroare:

```
Lista 3.3.6 din Shape.elm (safeHeronEnum)
                                                                                                                     Codul ulmului
 99 safeHeronEnum : Float -> Float -> Float -> Rezultat InvalidTriangleError Float
100 safeHeronEnum a bc =
101
         dacă (a < 0) atunci
102
            Err (partea negativă A)
103
         altfel dacă (b < 0) atunci
104
            Err (partea negativă B)
105
         altfel dacă (c < 0) atunci
106
            Err (NegativeSide C)
107
         altfel dacă ((a + b < c) | | (a + c < b) | | (b + c < a)) atunci
108
            Err ImpossibleTriangle
109
         altfel Ok (heron a bc)
```

```
Elm REPL

> safeHeronEnum 1 1 1 0k

0.4330127018922193 : Rezultat InvalidTriangleError Float > safeHeronEnum 1 2 1
Ok 0 : Rezultat InvalidTriangleError Float > safeHeronEnum 1 -1 3

Err (NegativeSide B) : Rezultat InvalidTriangleError Float

> safeHeronEnum 1 3 1

Err ImpossibleTriangle : Rezultat InvalidTriangleError Float
```

În ambele cazuri, funcțiile care returnează erori pot fi compuse cu ușurință:

În cazul erorilor de șir, putem returna eroarea fără nicio modificare sau "adăugăm" un context suplimentar la șir.

În cazul erorilor enumerate, putem include tipul de eroare al funcției apelate într-o variantă a tipului de eroare al funcției apelant. (de exemplu, în cazul safeHeronEnum , includem InvalidTriangleError cu va<mark>rianta InvalidTrian</mark>gle)

Întrebarea 3.3.4

**

Discutați cel puțin 2 avantaje și dezavantaje ale fiecărei abordări (eroare șir, variantă separată pentru fiecare eroare). În ce cazuri ați folosi unul față de celălalt?

3.4 Liste - partea 1

În Elm (precum și în alte limbaje moderne de programare funcțională) listele sunt liste legate individual, care sunt definite ca:

```
Lista 3.4.1: Definirea listei

tip List a
= Contra a (Lista a)
| Zero
```

Cu definiția de mai sus, putem crea liste după cum urmează:

```
> Nul
Nil: Lista a
> Contra 1 Nil
Contra 1 Nil : Numărul listei
> Contra 1 (Contra 2 Nil)
Cons 1 (Cons 2 Nil) : Numărul listei
```

Putem folosi constructorii pentru a construi liste:

```
Elm REPL

> countFromTo a b = if a >= b then Nil else Cons a (countFromTo (a+1) b) <function> : number -> number

-> List number > countFromTo 1 5

Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

și pentru a procesa liste folosind potrivirea modelelor:

```
Elm REPL

> sumOfElements I = | cazul

I de |

Nil -> 0 |

Contra x xs -> x + sumOfElements xs |

<funcţie>: Număr listă -> număr

> sumOfElements (countFromTo 1 10)

45 : număr
```

După cum puteți vedea, atât definirea, cât și tipărirea listelor în această formă este destul de pronunțată și greu de înțeles. Astfel, Elm (precum multe (nu doar) limbaje de programare funcțională) are un nivel de sintaxă pentru lucrul cu liste.

Conceptul 3.4.1: Sintaxă zahăr

Multe limbaje de programare oferă caracteristici care ajută la scrierea modelelor comune într-o manieră concisă. S-ar putea spune că sintaxa rezultată pentru realizarea acestor sarcini este "dulce și scurtă", de unde și denumirea de sintaxă zahăr.

Procesul prin care compilatorul procesează aceste constructe se numește deszugaring, o transformare de la forma concisă într-o formă care folosește doar câteva construcții primitive. structuri.

Exemplele includ supraîncărcarea operatorului în C++ și expresiile lambda în Java.

În primul rând, putem defini literali de listă între paranteze:

```
Elm REPL > [1, 2, 3] [1,2,3] : Numărul listei
```

În al doilea rând, avem op<mark>eratoru</mark>l (::) (citit ca "contra") pentru a construi liste într-un mod mai lizibil:

```
Elm REPL

> (::)

<funcție>: a -> Lista a -> Lista a

> 1 :: 2 :: 3 :: []

[1,2,3] : Numărul listei
```

În cele din urmă, puteți vedea mai sus că Nil poate fi înlocuit și cu literalul listă goală [] .

Deci, funcțiile definite mai sus ar arăta în mod normal astfel:

```
Codul ulmului
     Lista 3.4.2 din Lists.elm (countFromTo, sumOfElements)
 4 countFromTo: Int -> Int -> List Int
 5 countFromTo de la la =
        dacă de la >= până atunci
 6
 7
 8
        altfel
           de la :: countFromTo (de la + 1) la
13 sumOfElements: List Int -> Int
14 sumOfElements I =
       cazul I de
16
          [] -> 0
17
          x::xs -> x + sumOfElements xs
```

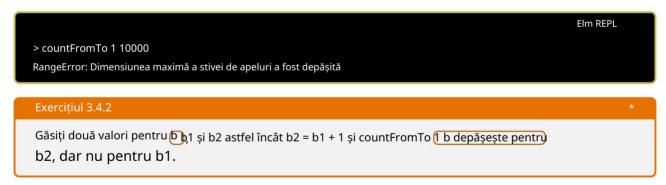
Exercițiul 3.4.1

Scrieți o funcție len ca<mark>le ret</mark>urnează acea lungime a unei liste (adică, numărul de elemente din

3.4.1 Lucrul cu liste în mod eficient

Întrebarea 3.4.1 Care este complexitatea de timp a următoarelor operațiuni dintr-o listă unică legată: 1. Introduceți la începutul listei (capul) 2. Inserați la capătul listei (coada) 3. Obțineți al- lea element

Unul dintre aspectele discutate în 1.9 la pagina 15 este că funcțiile recursive pot rămâne fără probleme. stiva spațiu dacă nu sunt scrise într-un stil recursiv de coadă.



Înainte de a ne uita la soluție, va fi util să aruncăm o privire la cum să inversăm listele:

```
Listarea 3.4.4 din Lists.elm (invers)
                                                                                                                      Codul ulmului
42 invers: Lista a -> Lista a
43 invers I =
44
        lăsa
45
           reverseAcc lx acc =
              cazul lx al
46
47
                  [] -> conform
48
                 x::xs -> reverseAcc xs (x::acc)
49
50
           reverseAcc I []
```

Aici putem observa că parametrul acumulatorului acc din funcția auxiliară reverseAcc acționează ca o stivă: de fiecare dată când funcția se autoinvocă recursiv, adaugă un element în partea de sus

Deci, ținând cont de faptul că, atunci când construim o listă într-o funcție recursivă de coadă cu acumulatori, acea listă va fi inversată, putem număra inversă în funcția auxiliară pentru a obține aceeași ieșire ca și funcția countFromTo):

```
Lista 3.4.5 din Lists.elm (countFromToTail)
                                                                                                                     Codul ulmului
21 countFromToTail: Int -> Int -> List Int
22 countFromToTail de la la =
23
        lăsa
           cnt a b acc =
24
              dacă a >= b atunci
25
26
                 conform
27
              altfel
28
                 cnt a (b - 1) ((b - 1)::acc)
29
30
           cnt de la la []
```



Nota 3.4.1

Pentru a lucra eficient cu liste legate individual, metoda preferată de a construi liste atașează elemente în fața listei.

3.4.2 Adăugarea listelor

Dacă vrem să facem operații mai complexe, avem nevoie în mod clar de mai multe abstracții lucrează cu liste.

Funcția care va fi foarte utilă este funcția append :



Aici trebuie să parcurgem prima listă pentru a accesa marcatorul de listă gol ([]) și a înlocui aceasta cu a doua listă.

De asemenea, putem rescrie acest lucru într-un stil recursiv de coadă, dar apoi trebuie să inversăm prima listă înainte atașarea fiecăruia dintre elementele sale la a doua listă, deoarece, așa cum am observat, construcția listei recursiunea în coadă funcționează într-un mod similar cu o stivă.

```
Lista 3.4.7 din Lists.elm (appendTail)
                                                                                                                         Codul ulmului
54 appendTail : Listă a -> Listă a -> Listă a
55 appendTail la lb =
56 lit
57
            appTail lx acc =
58
              cazul lx al
59
                  [] -> conform
60
                 x::xs -> appTail xs (x::acc)
61
        în
62
           appTail (invers la) lb
```

Întrebarea 3.4.2 * Care este complexitatea algoritmică a funcției append Tail ?

Avem și un operator pentru adăugarea listelor: (++):

```
Elm REPL
> ["Ave", "a"] ++ ["frumos", "zi"]
["Have","a","frumoasa","zi"] : Listă șir
```

Din nou pentru a înțelege importanța funcțiilor recursive de coadă, mai ales atunci când sunt operate liste, luați în considerare următoarea sarcină: adăugați două liste care conțin un interval de numere întregi și returnați lungimea acestuia (pentru a evita tipărirea listelor uriașe).

După cum puteți vedea atunci când folosim versiunile simple (non tail recursive) ale funcțiilor pe care le vor folosi debordează stiva.

Putem vedea indiferent de modul în care combinăm aceste funcții, dacă una dintre ele nu este recursivă vor deborda stiva.

```
Elm REPL

> len (countFromTo 0 10000)

RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită

> lenTail (countFromTo 0 10000)

RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită

> lenTail (countFromToTail 0 10000)

10000: număr
```

```
Elm REPL

> lasa

| 11 = countFromToTail 0 10000
| 12 = countFromToTail 20000 30000
| în

| lenTail (anexează l1 l2)
|
RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită
> lasa

| 11 = countFromToTail 0 10000
| 12 = countFromToTail 20000 30000
| | | în
| lenTail (appendTail 11 l2)
|
20000 : număr
```

3.4.3 Funcțiile capului și cozii

Fiecare implementare decentă a listei conectate oferă o funcție care returnează primul element al lista și o funcție care returnează lista fără primul element. In Elm aceste functii

se numesc cap si coada :



Acestea sunt exemple reale de funcții care folosesc tipul Maybe pentru a semnala că funcția nu poate returnează un rezultat valid pentru toate intrările:

Care este primul element al unei liste goale?

Cum putem sări peste primul element al unei liste goale?

În ambele cazuri, modalitatea ușoară de ieșire este să returnezi Nimic .

3.5 Probleme de practică

