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 bază Elm	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) și 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) — Codul ulmului

```
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) — Codul ulmului

```
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) — Codul ulmului

```
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
```

Nota 3.1.1

Variabilele de tip și parametrii funcției pot avea același nume, dar asta nu înseamnă că sunt legați în vreun fel!
Următoarele funcții sunt semantic aceleași:

Lista 3.1.5: makeTuple — Codul ulmului

```
makeTuple : a -> b -> (a, b) makeTuple
a b = (a, b)
--

makeTuple : fst -> snd -> (fst, snd) makeTuple a
b = (a, b)
--

makeTuple : b -> a -> (b, a) makeTuple
a b = (a, b)
--

makeTuple : x -> y -> (x, y) makeTuple
elem1 elem2 = (elem1, elem2)
```

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, cum , (>) , 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 parAdd 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 și List

number : tipuri care sunt numere: Int și Float

comparabil : tipuri care pot fi comparate: numere, Char și tuplu de , , 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.

Întrebarea 3.2.1 ***

Cum se gestionează egalitatea pentru tipul Float ? Încercați să evaluați în REPL: (0/0) și apoi (0/0) == (0/0) . Cauza acest lucru o problemă pentru egalitatea de referință?

Sugestie: Gândiți-vă la cazul în care aveți un tip care conține un câmp de tipul Float .

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

Întrebarea 3.3.1 **

Ce este cunoscut sub numele de „greșeală de un miliard de dolari” în informatică?

Conceptul 3.3.1: Anulabilitate

În limbaje de programare precum C, C++ și Java, folosim valoarea nulă pentru a indica faptul că un pointer sau o referință este invalidă.

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 capitolul 2. Funcția nu este 100% corectă, pentru că nu numai că o putem numi cu argumente care reprezintă trei laturi care nu pot.

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 — Codul ulmului

```
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 tipul de returnare la Maybe Float și returnând Nimic dacă argumentele sunt negative sau nu pot forma un triunghi valid:

Lista 3.3.2 din Shape.elm (validTriangle, safeHeron) — Codul ulmului

```
39 triunghi valid a bc = 40 ((a > 0)
&& (b > 0) && (c > 0)) && 41 (a + b >= c) && (b + c) >= a)) 45
safeHeron : Float -> Float -> Float -> Maybe Float 46
    safeHeron a bc = 47 48 49 50


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

```
Elm REPL

> import Form expunerea (..) >
safeHeron 1 1 1 Doar
0,4330127018922193 : Poate că
safeHeron 2 2 10 Nimic : poate că
safeHeron -2 3 4

Nimic: Poate Plutește
```

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?

Întrebarea 3.3.3 ***

Cunoașteți vreo limbă care are o soluție în timp de compilare pentru problema anulării încorporată ca caracteristică de limbă?
Oferiți o implementare a funcției safeHeron într-un astfel de limbaj.
Comparați utilizarea tipurilor de sumă pentru a reprezenta tipurile nullabile (adică nu sunt tratate într-un mod special de limbaj) cu această abordare (adică conceptul este încorporat în limbaj).
Care sunt, în opinia dumneavoastră, principalele avantaje și dezavantaje ale fiecărei abordări?
Pe care o preferați?

Notă:

„Timp de compilare” înseamnă că compilatorul va emite cel puțin un avertisment dacă un câmp sau o variabilă ar putea fi nulă (și, prin urmare, ar cauza probleme atunci când este anulată de referință în timpul rulării).
„Funcția încorporată ca limbaj” înseamnă că există o sintaxă specială pentru aceasta. În Elm nu există o sintaxă specială, deoarece anulabilitatea este exprimată de tipul Mabye , care este definit în biblioteca standard. Ca un indiciu suplimentar, Rust rezolvă problema nulității la fel ca Elm: nu are conceptul de null, iar tipul său pentru a semnala anulabilitatea este definit ca un tip de sumă numit Opțiune (deci nu satisface acest criteriu).

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 folosi tipul de rezultat , care poate include un rezultat de succes în Varianta Ok și un tip de eroare în varianta Err :

Lista 3.3.3: Definiția tipului de rezultat — codul ulmului

```
tip Valoare eroare rezultat
    = Ok valoare
    | Err eroare
```

Un caz de utilizare bun pentru acest tip ar fi funcția de zonă , 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) — Codul ulmului

```
19 safeArea : Shape -> Result String Float
20 safeArea shape =
21 forma carcasei de
22         Raza cercului ->
23             dacă raza < 0 atunci
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             altfel
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 Err .

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.

Elm REPL

```
> safeArea (Triunghi 2 2 3)
Ok 1.984313483298443 : Rezultat șir 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) — Codul ulmului

```
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 Ok
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 varianta InvalidTriangle)

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 Codul ulmului

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

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

Elm REPL

```
> 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 l = | cazul
l de |
            Nil -> 0 |
            Contra x xs -> x + sumOfElements xs |

<funcție>: 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 operatorul (::) (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:

Lista 3.4.2 din Lists.elm (countFromTo, sumOfElements) — Codul ulmului

```
countFromTo : Int -> Int -> List Int
countFromTo de la la =
    dacă de la >= până atunci
        []
    altfel
        de la :: countFromTo (de la + 1) la
sumOfElements : List Int -> Int
sumOfElements l =
    cazul l de
        [] -> 0
        x::xs -> x + sumOfElements xs
```

Exercițiul 3.4.1 *

Scrieți o funcție len care returnează acea lungime a unei liste (adică, numărul de elemente din aceasta).

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.

Elm REPL

```
> countFromTo 1 10000
RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită
```

Exercițiul 3.4.2 *

Găsiți două valori pentru b b1 și b2 astfel încât b2 = b1 + 1 și countFromTo 1 b depășește pentru b2, dar nu pentru b1.

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

```
invers: Lista a -> Lista a
invers l =
    lăsa
        reverseAcc lx acc =
            cazul lx al
                [] -> conform
                x::xs -> reverseAcc xs (x::acc)
    în
        reverseAcc l []
```

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 aceasta.

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
24         cnt a b acc =
25             dacă a >= b atunci
26                 conform
27             altfel
28                 cnt a (b - 1) ((b - 1)::acc)
29     în
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 :

Lista 3.4.6 din Lists.elm (anexează) Codul ulmului

```
34 append : Lista a -> Lista a -> Lista a
35 anexează lx ly =
36 caz lx de
37         [] -> ly
38         x::xs -> x :: anexează xs ly
```

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 appendTail ?

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).

Elm REPL

```
> lasa
        l1 = countFromTo 0 10000
        l2 = countFromTo 20000 30000
| | | în
| len (adăugați l1 l2)
|
RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită
```

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
 | l1 = countFromToTail 0 10000
 | l2 = countFromToTail 20000 30000
 | în
 | lenTail (anexează l1 l2)
 |
RangeError: Dimensiunea maximă a stivei de apeluri a fost depășită
> lasa
        l1 = countFromToTail 0 10000
        l2 = countFromToTail 20000 30000
 | | | în
 | lenTail (appendTail l1 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 :

Lista 3.4.8 din Lists.elm (cap, coadă) — Codul ulmului

```
66 cap : Lista a -> Poate a
67 cap l =
68 cazul l de
69             [] -> Nimic
70 x::_ -> Doar x

74 coada : Lista a -> Poate (Lista a)
75 coada l =
76         cazul l de
77             [] -> Nimic
78             _::xs -> Doar xs
```

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

Exercițiul 3.5.1 *

Scrieți o funcție cu semnătura safeDiv : Int -> Int -> Maybe Int care revine Nimic atunci când încercăm să împărțim la 0 și rezultatul înfășurat în Doar altfel.

Exercițiul 3.5.2 *

Rescrieți funcția len definită mai sus într-un stil recursiv de coadă, cu numele lenTail .

Exercițiul 3.5.3 *

Implementați o funcție ultima , care returnează ultimul element al unei liste.

Exercițiul 3.5.4 *

Scrieți o funcție indexList il , th care returnează i element al listei l .

Exercițiul 3.5.5 **

Scrieți o funcție fibs start end o listă a , care ia două numere, început și sfârșit și se întoarce numerelor Fibonacci astfel încât:

f ibs(start, end) = {f ib(i) |i N, start i < end}

Elm REPL

```
> fibs 0 1
[1]: Numărul listei
> fibs 0 2
[1,1] : Numărul listei
> fibs 0 3
[1,1,2] : Numărul listei
> fibs 0 4
[1,1,2,3] : Numărul listei
> fibs 3 10
[3,5,8,13,21,34,55] : Număr listă
```

Exercițiul 3.5.6 **

Modificați funcția fibs pentru a returna o listă de tupluri, unde primul element din fiecare tuplu denotă indicele numărului Fibonacci și al doilea numărul Fibonacci însuși.

Elm REPL

```
> fibs 0 2
[(0,1),(1,1)] : Listă ( număr1, număr )
> fibs 0 3
[(0,1),(1,1),(2,2)] : Lista ( număr1, număr )
> fibs 0 4
[(0,1),(1,1),(2,2),(3,3)] : Listă ( număr1, număr )
> fibs 3 10
[(3,3),(4,5),(5,8),(6,13),(7,21),(8,34),(9,55)] : Lista ( număr1, număr )
```

Exercițiul 3.5.7 ***

Scrieți o funcție cu semnătura cmpShapes : Shape -> Shape -> Result String Order care folosește funcția safeArea pentru a calcula aria celor 2 forme de intrare și returnează ordonarea dintre ele înfășurată în varianta Ok , dacă aria ambelor forme poate fi calculat.
În caz contrar, ar trebui să returneze un mesaj de eroare împachetat în varianta Err .

Elm REPL

```
> cmpShapes (Cercul 2) (Cercul 3)
Ok LT : Ordinea șirurilor de
> cmpShapes (Triunghi 2 2 2) (Cercul 3)
Ok LT : Ordinea șirurilor de
> cmpShapes (Dreptunghi 4 4) (Cercul 3)
Ok LT : Ordinea șirurilor de
> cmpShapes (Dreptunghi 6 6) (Cercul 3)
Ok GT : Ordinea șirurilor de
        > cmpShapes (Dreptunghi 2 3) (Triunghi 3 4 5)
Ok EQ: Ordinea șirurilor de
> cmpShapes (Dreptunghi 2 3) (Triunghi -3 4 5)
Err ("Introducere nevalidă pentru forma dreaptă: laturile nu pot forma un triunghi")
      : Ordinea șirurilor de
> cmpForme (Dreptunghi -2 3) (Triunghi -3 4 5)
Err ("Introducere nevalidă pentru forma din stânga: lățime sau înălțime dreptunghi negativă")
      : Ordinea șirurilor de
```

Exercițiul 3.5.8 ***

Scrieți o funcție cu semnătura: totalArea : List Shape -> Result (Int, InvalidShapeError) Float care utilizează funcția safeAreaEnum pentru a calcula suprafața totală a formelor de intrare din listă. Ar trebui să returneze suprafața totală (suma tuturor suprafețelor) în varianta Ok , dacă toate zonele pot fi calculate. În caz contrar, ar trebui să returneze un tuplu cu eroarea returnată de safeAreaEnum și indexul formei care a cauzat eroarea încapsulat în varianta Err .

Elm REPL

```
> totalArea [Cercul 2, Cercul 2]
Ok 25.132741228718345 : Rezultat ( Int, InvalidShapeError ) Float > totalArea [Cerc
2, Cerc 2, Dreptunghi 3 4, Triunghi 3 4 5]
Ok 43.132741228718345 : Rezultat ( Int, InvalidShapeError ) Float > totalArea [Cerc
2, Cerc 2, Dreptunghi 3 4, Triunghi 1 1 5]
Err (3,InvalidTriangle ImpossibleTriangle): Rezultat ( Int, InvalidShapeError ) Float > totalArea [Cerc 2, Cerc
-2, Dreptunghi 3 4, Triunghi 1 1 5]
Err (1,InvalidCircle): Rezultat ( Int, InvalidShapeError ) Float
```