cd Haskell-For-Dummies/

stack repl

HASKELL

- LAMBDA
  calculus programlari ifade edebilmek icin bir hesaplama veya muhakeme yontemidir. efektif hesaplama bicimi icin bir prosestir lambda. Uc temel terms icerir. expressions, variables(potantial inputs),abstractions.

What is functional programming?
fonksiyonlarin matematiksel fonksiyonlara dayandirildigi bir hesaplama paradigmasidir. Fuctional programming daha cok soyutlama yapabilmenizi saglar boylece daha kisa kod yazarak yapacaginiz isi yapabilirsiniz. Ifadeleri bir deger olarak degistirip programa davranisini degistirmeden dogru calismasini saglamak fonksiyonel programlamanin temel fikridir.

Fonksiyon nedir?
Bir fonksiyon girdi ile cikti arasindaki baglantidir.bir takim ifadelerden olusan ve bir input tanimlandiginda buna bir sonuc veren veya verilen inputu reduce eden… Bir fonksiyonun birden fazla girdisi olabilir ama birden fazla ciktisi olmaz.

lambda x.x --->head, parameter(variable), body
(lambda x.x) (lambda y.y) ---->[x:= (lambda y.y)] sagdaki kismi sola input olarak atiyoruz. bu kurala left associative deniyor.
(lambda xy.xy) = (lambda x.(lambda y.xy))
(lambda xy.xy) (lambda z.a) 1 = (lambda x.(lambda y.xy)) (lambda z.a) 1 = (lambda y.((lambda z.a) y) 1 = (lambda z.a)1 = a
lambda x.xy ifadesinde y free variable iken x bound variable.
Combinator?
free variable icermeyen bir lambda terimidir. ornegin lambda x.x gibi. lambda xy.x de combinator dur. x de y de free degil cunku. ama lambda y.x bir combinator degildir. cunku x free variable dir.

*Divergence?
Daha kuculmuyor hatta buyuyor boyle durumlar divergence oluyor. ornegin (lam x.xx)(lam x.xx)

REPL =read eval print loop(programlama icin kullandigin environment)

*To exit from ghci is :q command
*you can work from source file or repl.

*:: bu type signature vermeye yariyor.

yazdigin bir .hs dosyasini repl a yuklemek icin :load test.hs yaz
yukledigin dosyadan cikip >prelude a geri donmek icin :m yaz
fonksiyon isimleri kucuk harfle baslamali ve tek harf olmamali
source file dan degisiklik yaptin o zaman repl da calisitigini gormek icin :r yapmalisin.
infix style yani sembol ile fonksiyon--> 10 `div` 4 = div 10 4 = 2
(+) 100 10 = 110
:info (*) carpma icin type hakkinda bilgi verir
2^3^4 = 2^(3^4) tur. cunku left associative.
let x = 5
her .hs dosyasinin basina module dosya-adi where yazmak gerekiyor.
let x=3 y=4 veya let x=3 y=4 calismaz. y yi yazdigin kisim biraz daha sagda olmali. bu ifade hatali calisiyor, alttaki ornekte denedim

```
foo x =
    let y = x * 2
        z = x ^ 2
    in 2 * y * z
```

fonksiyonu yazarken basta bosluk birakmadan yazmalisin. yoksa calismaz.

1/1 = 1 iken div 1 1 = 1.0
mod 1 1 = 0 rem 1 1 = 0 quot 1 1 = 1
div ile quot yuvarlama yapiyor.
rem returns the dividend, if the dividend is less than the divisor. div rounds off the divisor to the negative infinity, if either the dividend or divisor is negative. **mod** follows the sign of the divisor. div returns 0 , if the dividend is less than the divisor, and both arguments are positive.
div 20 (-6) = -4 bir yukariya yuvarlarken quot bir asagiya yuvarliyor.
quot 20 (-6) = -3

mod (-9) 7= 5 rem (-9) 7 = -2 mod bir yukariya yuvarlarken rem bir asagiya yuvarliyor. yani div ile mod yukariya yuvarlama yapiyor.

(-5) `rem` (-2) = -1

:info (*) dersek carpi hakkinda bilgi almis oluruz.

$ isareti islem yaparken parantez ile ugrastirmaz seni.
ornegin: (2^) $ 2 + 2 dersen (2+2) demene gerek kalmaz.

(2^) $ (+2) $ 3 * 2 = 256

(2^) $ 2+2 $ (*30) calismaz.

1 + 2 = (+) 1 2 = (+1) 2 = 3

(subtrack 2) 3 = 1

```
printInc n = print plusTwo
  where plusTwo = n + 2
```

```
printInc2 n = let plusTwo = n + 2
              in print plusTwo
```

```
-- this should work in GHCi
let x = 5; y = 6 in x * y
```

could be rewritten as

```
-- put this in a file
mult1      = x * y
   where x = 5
         y = 6
```

let veya where ile yazdigin sadece o fonksiyon icerisinde calisir.

```
let x = 7
    y = negate x
    z = y * 10
in z / x + y
```

bir seyin type ini ogrenmek istiyorsan :type ‘a’ ornegin a karakterinin type ini ogrenmis oluruz.

:type “hello” string type ini sormus olursun.

print “hello world”

putStrln “hello world” ile yukaridaki ayni ama putS... string icin sadece kullanilabilirken print her sey icin kullanilir.

```
main :: IO ()
main = putStrLn "hello world!"
world :: String
world = "world!"
```

eger kendi prompt unu degistirmek istiyorsan :set prompt “ne istiyorsan” yaz.

IO = ekrana yazdirmak istedigimiz sey gibi dusun. Mesela putStrln “hello” dersek string i alir ve ekrana IO olarak gösterir.

Concatenate ?

bir seyleri birbirine bagli etmek. ornegin hello world yazdirmak istiyorsun. bunu yaparken hello yu bir fonksiyon, world u de bir fonksiyon gibi yazip birbirine link ederek hello world yazdirabilirsin.

:set prompt "deniz kod yaziyor>" prelude un ismini degistirir

['a'] ++ ['d'] = “ad”

[Char] = list of characters

++ bir infix operatordur. basta kullanilirsa parantez gereklidir. ama concat oyle degil

concat [[1,2], [3,3,4], [6,7]] = [1,2,3,3,4,6,7]

concat [“deniz”, “pinar”, “akillidir”] = denizpinarakillidir

“hello” ++ “chris” = hellochris

(++) “hello” “world” = helloworld

head “deniz” = ‘d’

tail “deniz” = “eniz”

take 1 “deniz” = “d”

drop 2 “deniz” = “niz”

“deniz” !! 0 = ‘d’

“deniz” !! 4 = ‘z’

CHAPTER 4

Datatypes?

restricting the forms of data our programs can process. datatype bize bir aractir daha hizli programlama icin ve de maintanence icin buyuk avantaj saglar. strings, characters, numbers birer datatype tir.

Type constructer?

senin kod icerisinde ne tur komut veya fonksiyon kullandigindir. boolean logic bir type cons tur.

Data constructer?

senin kodun icinde kullandigin degerlerdir. ya da senin kodunun evaluate ettigi degerlerdir. ornegin logical islem yapiyorsan true ve false senin data cons. larindir.

Kendi dataype larini yapmak istediginde deriving show bunu ekrana yansitmani saglar. data Mood = Blah | Woot deriving Show

Fractional numeric types?

float: devir gibi uzunca devam eden sayilarin gosterimi
double :
rational : numerator and denominator den olusur.
scientific :

Int type inin dezavantaji buyuk ardisik bilgi miktarini aciklayamamasi.

127 :: Int8 dersem sonuc olarark 127 donduruyor. ama ardindan 128::Int8 dersem error verecek.

minBound :: Int8 = -128

minBound :: Int16 = -32768 aslinda 2 s compement iifade ediyor. binary olarak

maxBound :: Int8 = 127

maxBound :: Int16 = 32767

FRACTIONAL NUMBER

fractional sayilar bir typeclass tir.

4/2 diye sorarsak repl a bize kesirli olarak cevap verir. 2.0 diye.

let x=5 ,x==5 dersek sonuc olarak true dondurur.

let x=5, x/=5 dersek bize false dondurur. cunku /= isareti esit degildir degil mi? demek aslinda

true ve false birer data constructer lardir bool datatype icin.

Eq a => a -> a -> Bool type aciklamasinda EQ bir typeclass tir.

(/) :: Fractional a => a -> a -> a burada gordugun kucuk esittir ifadesi input olan a lardan birinin fractional bir sayi ile sinirlandirilmis oldugunu gosterir.

y =5 dedikten sonra y == 5 dersen true sonucunu verir. == equal? demektir aslinda. Pekiyi not equal? i nasil sorariz? /= ifadesi ile. Yani y /= 5 dersem false cevabini alirim cunku y=5 tir.

Ord a => a -> a -> a -> Bool ifadesindeki Ord her cesit ifadeyi iceren bir typeclassstir.

repl a sunu yazarsan true cevabini verir. 'a'<'b'

"JULIE" >"CHRIS" yazarsak sonuc true olur. j>c cunku. eger esitse bir sonrakine gecer ve kiyaslar ve boylece devam eder.

true && true yazarsak true esitmidir true ya? demis oluruz.

ornegin (8>4) && (4>5) dersek sonuc false. yani iki tane && isareti and? yerine geciyorken || bu isaret or? yerine geciyor.

(8>4) || (4>5) yazarsak sonuc true.

haskell if komutuna sahip degil. onun yerine bool datatype ini kullanabiliriz.

if true then "deniz" else "pinar" ifadesinin sonucu ="deniz" (aslinda bu su demek eger true esittir true ise sonuc deniz degilse pinar olsun.

if false then "deniz" else "pinar" dersek sonuc "pinar" olur. cunku eger false esittir true ise sonuc deniz degilse sonuc pinar olsun demis oluyoruz.

```haskell
-- greetIfCool1.hs
module GreetIfCool1 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool =
          coolness == "downright frosty yo"
```

When you test this in the REPL, it should play out like this:

```
Prelude> :l greetIfCool1.hs
[1 of 1] Compiling GreetIfCool1
Ok, modules loaded: GreetIfCool1.
Prelude> greetIfCool "downright frosty yo"
eyyyyy. What's shakin'?
Prelude> greetIfCool "please love me"
pshhhh.
```

TUPLES?
tuple oyle bir cesit ki sana tekil bir deger icindeki coklu degerler arasinda gecis yapmani veya store yapmani saglar. tuple = tanimlama grubu
mesela (x,y) de iki tane tuple varken (x,y,z) de uclu tuple var.
(,) 8 “luicc” dersem (8, “luicc”) sonucunu verir.

```
Prelude> let myTup = (1 :: Integer, "blah")
Prelude> :t myTup
myTup :: (Integer, [Char])
Prelude> fst myTup
1
Prelude> snd myTup
"blah"
Prelude> import Data.Tuple
Prelude> swap myTup
("blah",1)
```

We had to import `Data.Tuple` because `swap` isn't included in the `Prelude`.

We can also combine tuples with other expressions:

```
Prelude> 2 + fst (1, 2)
3
Prelude> 2 + snd (1, 2)
4
```

listeler tek bir deger icerisindeki coklu degerleri iceren baska bir type tir.

tuple ile list arasindaki uc fark nedir?

listedeki tum elementler ayni type ta olmali, listelerin kendi syntaxlari var, degerlerin sayisi belli olmaz listelerin type kisminda.

awesome = ["papuchon", "curry", ":)"] yazdiktan sonra length awesome nedir dersek sonucu 3 olarak dondurur.

length [1, 'a', 3, 'b'] bu calismaz.

reverse "blah" = "halb"

A tuple is an ordered grouping of values.

In Haskell, you cannot have a tuple with only one element, but there is a zero tuple also called unit or ().

Type constructors in Haskell are not values and can only be used in type signatures.

data Pet = Cat | Dog Name

data Deniz = Iyi | Kotu Davranis

In the above example, Pet is the type constructor. A guideline for differentiating the two kinds of constructors is that type constructors always go to the left of the = in a data declaration.

Data declarations define new datatypes in Haskell. Data declarations always create a new type constructor, but may or may not create new data constructors. Data declarations are how we refer to the entire definition that begins with the data keyword.

In Haskell there are seven categories of entities that have names: functions, term-level variables, data constructors, type variables, type constructors, typeclasses, and modules.

Term level kodunun calistigi (canli) seviyedir.

Type level ise kodunun yazildigi tanimlandigi seviyedir.

Neden bizim type a ihtiyacimiz var? cunku matematiksel islemler surekli dogrulugu zorlar ve birseyleri daha cok manipule etmenizi zorlasitirir, ama type ile biz manipule etmek istedigimiz seyi programa bildirerek gereken degisiklik icin izin almis oluruz.

The arrow, (->), is the type constructor for functions in Haskell

Prelude> :type length

length :: [a] -> Int buradaki [a] bir argumanli bir listeyi ifade ediyor.

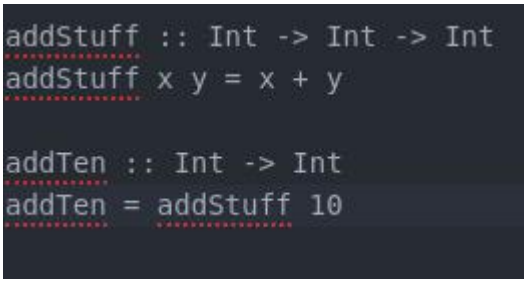
all functions in Haskell take one argument and return one result. Other programming languages, if you have any experience with them, typically allow you to define functions that can take multiple arguments. There is no support for this built into Haskell. bunun yerine cok parametreli fonksiyonlari icin haskell da currying metodu uygulanir, her biri bir arguman alir ve bir sonuc donduren taktiksel kolayliklar kullanilir. en distaki fonksiyon gelecek argumani kabul edebilmek icin baska bir fonksiyona donusuyor, bu yonteme currying denir

ASAGIDAKI ORNEKTE BIR FONKSIYON OLUSTURUP BU FONKSIYONU BASKA BIR FONKSIYONA DONUSTUREREK HALLEDIYORUZ.

```
Prelude> :t addStuff
addStuff :: Integer -> Integer -> Integer
Prelude> let addTen = addStuff 5
Prelude> :t addTen
addTen :: Integer -> Integer
Prelude> let fifteen = addTen 5
Prelude> fifteen
15
Prelude> addTen 15
25
Prelude> addStuff 5 5
15
```

```
subtractStuff :: Integer
              -> Integer
              -> Integer
subtractStuff x y = x - y - 10
subtractOne = subtractStuff 1

Prelude> :t subtractOne
subtractOne :: Integer -> Integer
Prelude> let result = subtractOne 11
Prelude> result
-20
```



• Uncurried functions: One function, many arguments
• Curried functions: Many functions, one argument apiece
anonymous = \i b -> i + (nonsense b) buradaki \ isareti lambda gibi gorev yapiyor.
yani (lambda i b) = i + (nonsense b) gibi.

```
Prelude> let curry f a b = f (a, b)
Prelude> fst (1, 2)
1
Prelude> curry fst 1 2
1
```

burada uncurry olan bir fonksiyonu curry yapiyor

```
let uncurry f (a, b) = f a b
Prelude> (+) 1 2
3
Prelude> uncurry (+) (1, 2)
3
```

burada da curry olan bir bir fonksiyonu uncurry yapiyor.

sectioning = infix operatorlerin kismi uygulamasi, aslinda birden fazla fonksiyonu birlestirerek kullanmamizi ve daha genis soyutlama yetenekleri kazandirirarak sagliyor bunu. asagidaki c 25 ornegi gibi.

(^2) 5 = 25

Prelude> elem 9 [1..10]
True
Prelude> 9 `elem` [1..10]
True
Prelude> let c = (`elem` [1..10])
Prelude> c 9
True
Prelude> c 25
False

polymorfik demek bir cok bicimden bir araya gelmis demek
polimorfik degisken turleri bize ifadeleri arguman kabul ederek soyutlama yetenegini bize verir ve farkli turlerin varyasyonlarini her seferinde yazmadan sonucunu dondurmemizi saglar

Broadly speaking,
type signatures may have three kinds of types: concrete, con-
strained polymorphic, or parametrically polymorphic.
In Haskell, polymorphism divides into two categories: para-
metric polymorphism and constrained polymorphism.

id = identity parametrically polymorfic function. aslinda su demektir bu;bir typeclass ile sinirlandirilmamis yani fully polymorfic demektir

f :: a -> a
f x = succ x
bunu yazarsan calismaz cunku a bir parametrik polymorfic yani sinir tanimaz tur olarak, ama succ ise bir sayi almali yani contrained polymorfic, bu nedenle error verir.

dp :: a -> a -> a
dp x y = x + y bu fonk calismaz. cunku iki tane a girdisi var ve sonuc yine a ama hangi a? belli degil.
SAYFA 217