Дәріс 3-4

Haskell **тілде типтерді анықтау**

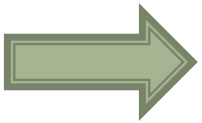
let- байланыстыру

$ax^2 + bx + c = 0$: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

- roots a b c =

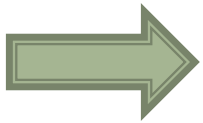
((-b + sqrt (b*b - 4*a*c)) / (2*a), (-b - sqrt (b*b - 4*a*c)) / (2*a))

```
roots a b c =
let det  =  sqrt    (b*b - 4*a*c)
in  ((-b+det /   (2*a),  (-b- det /   (2*a))
roots a b c =
let det = sqrt (b*b - 4*a*c)
      twice_a =2*a
in  ((-b + det)/ twice_a, (-b-det)/ twice_a)
```



- roots a b c =

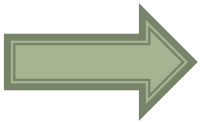
```
let { let = sqrt(b*b - 4*a*c); twice_a =2*a }
in  ((-b + det)    / twice_a,
(-b - det)    / twice_a)
```



. . where ... конструкциясы

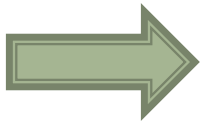
- roots a b c =

((-b + det) / twice_a,
(-b - det) / twice_a)
where det = sqrt (b*b - 4*a*c)
twice_a = 2 *a



Глобалды функциялардың қолданалуы

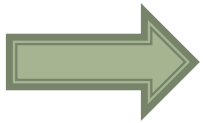
```
det  a b c= sqrt    (b*b - 4*a*c)
        twice_a  a =   2 *a
        roots a b c =
        ((-b + det a b c)   / twice_a  a,
        (-b-det a b c)   / twice_a  a)
```



Let және where конструкцияларда функцияларды пайдалану

Берілген m санынан натуралды сандар тізімін [m,m+1, m+2,…, n] қайтаратын numsFro көмекші функцияны енгіземіз және осы анықтаманы локалды етеміз

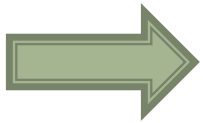
```
numsTo n=
let numsFrom m= if m ==  n then [m]   else
  m:numsFro (m+1)
in numsFrom 1
```



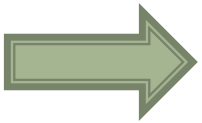
Қателіктер туралы хабарлар

factorial 0 = 1

factorial n = n* factorial (n-1)

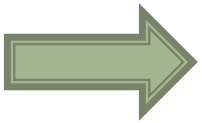


```
factorial 0 = 1
factorial n = if  n>  0  then
n* factorial  (n-1)
else
error "factorial:  negative    argument"
```



Сақтаушы шарттар

```
factorial 0 = 1
factorial n | n<  0  =  error  “factorial : negative argument”
            | n>=  0  =  n * factorial (n-1)
```



Санның таңбасын анықтайтын функция

```
signum x   |   x <  0     =  -1
           | x == 0       =    0
           | otherwise = 1
```

Шартты операторды пайдалануда **signum** функцияны анықтау

```
signum x  = if x <0      then
       -1
       else
if x == 0      then
    0
else
-1
```

Полиморфты типтер

Haskell тілінде типтердің полиморфты жүйесі қолданылады. Белгілі тізімнің бірінші элементін қайтаратын tail функциясын қарастырайық

Prelude> tail [1,2 ,3]

[2,3]

Prelude> tail [‘a’,’b’, ‘c’]

[’b’, ‘c’]

Prelude> tail [“list”,”of”, “lists”]

[”of”, “lists”]

tail функциясы полиморфты тип: [a] -> [a]. Бұл оның аргумент ретінде кез келген тізімді қабылдап, сол типтің тізімін қайтаратынын білдіреді. Мұнда a типтік айнымалыны білдіреді, яғни оның орнына кез келген нақты типті қоюға болады.

a типтік айнымалыны білдіреді, яғни оның орнына кез келген нақты типті қоюға болады. Сонымен [a] -> [a] жазбасы типтердің бүтін бір жанұясын көрсетеді мысалы, [Integer] -> [Integer], [Char] -> [Char], [[Char]] -> [[Char]] және т.б.

Полиморфты типтерде айнымалылардың бірнеше типтерін қолануға мүмкін, мысалы fst функциясы (a,b) > a типті. Бұл типті анықтағанда екі типтік айнымалы қолданылады

Пайдаланушы типтері

Мысал ретінде стандарттыға ұқсас жұптың анықталуын қарастырамыз.

data Pair a b = Pair a b

data – кілттік сөзі типті өзіміз анықтайтымызды білдіреді;

a b- типтің параметрлерін білдіретін типтік айнымалылар.

Теңдік белгісінен кейін біз осы типтің берілгендер конструкторын көрсетеміз яғни Pair (берілгендер конструкторының атауы тип атауымен сәйкес келуі міндетті емес). Конструктор атауынан кейін біз тағы да a b жазамыз, бұл жұпты конструкциялау үшін бізге a және b типіне жататын екі мән қажет екенін білдіреді.

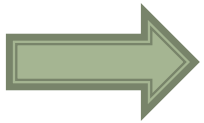
Бұл анықтама Pair : : a -> b -> Pair a b, функциясын енгізеді, ол Pair типті жұптарды конструкциялау үшін қажет.

Бұл кодты интерпретаторға енгізіп, жұптардың қалай конструкцияланатынын көруге болады.

```
 Main>:t  Pair
Pair  ::  a  ->  b  ->  Pair a b
Main> : t  Pair ‘a’
Pair  'a' ::  a  ->  Pair    Char a
Main>: t   Pair 'a'  "Hello"
Pair 'a'  "Hello"   ::  Pair    Char   [Char]
```

Берілгендер конструкторларына сәйкес функцияларды қолдану кезінде үлгімен сәйкестентіретін қасиеті бар. Осылайша бұл жұптың бірінші және екінші элементін алу үшін:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```



Көпшілік конструкторлар

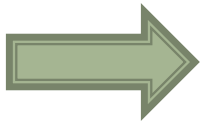
Өзіндік типтерді құруда бір- бірінен ' |' символымен ажыратылатын бірнеше конструкторларды қолдану мүмкін.

Түстерді ұсынатын Color типін қарастырайық, R Gee және Be мүмкін мәндері бар Оны былайша анықтауға болады

data Color = Red | Green | Blue

мұнда Color — тип атауы, ал Red, Green және Blue – берілгендер конструкторлары.

Назар аударыңыз, бұл тип параметрлер қабылдамайды. Бұл типтар санаушы деп аталады.

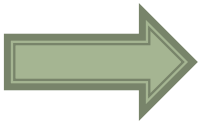


Алайда көпшілік конструкторларда параметрлер қабылдауы мүмкін. Осылайша Color типі тек үш түсті ғана анықтауға мүмкіндік беретінін байқауға болады. Оны үш бүтін санмен берілетін қызы, жасыл және көк түстердің деңгейіне сәйкес келетін (стандартты rgb) кез келген түсті анықтайтындай етіп кеңейтейік.

```
data      Color =  Red   |   Green   |   RGB Int      Int
   Int
```

Мұнда Color типі, стандартты түстерден басқа Red, Green және Blue, RGB конструкторының көмегімен үш бүтін санды қабылдап түстердің rgb-компоненттері бар кез келген түсті анықтауға мүмкіндік береді. Сонда, мысалы, түстің red-компонентін ерекшелеу былай жазылады

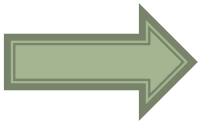
```
redComponent ::   Color   ->  Int
redComponent Red =   255
redComponent (RGB  r  _ _)  = r
redComponent  _  =  0
```



Көпшілік конструкторлы типтер полиморфты болуы мүмкін. Келесі мәселені қарастырайық. Функция қандай да бір нәтижені қайтаруы немесе полиморфты болуы мүмкін. Мысалы:

1) сызықтық теңдеуді шешу функциясы табылған түбірді қайтарады
2) тізімдегі бірінші теріс емес санды іздеп, осы санды қайтарады.

Сонымен бірге теңдеудің шешімі болмауы да мүмкін, тізімде теріс сандар болмауы мүмкін.

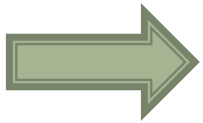


Бұл мәселе стандартты Maybe типімен шешіледі

```
Data Maybe      a = Nothing       |   Just a
```

Maybe типі (ағыл. maybe - мүмкін) типтік a айнымалысымен параметрленген және екі конструкторды ұсынады: Nothing (ағылш. ештеңе) шешімнің жоқтығын көрсету үшін және Just (ағылш. қарапайым, дәл) ойластырылған нәтиже. Онда біздің функцияны мына түрде жазуға болады:

```
-- Функция теңдеудің түбірін қайтарады ax + b =  0
solve   ::   Double  -> Double   - > Maybe Double
solve  0   b = Nothing
solve  a b = Just   (- b / a)
-- Функция тізімнің бірінші теріс емес элементін қайтарады
findPositive     ::   [Integer]       -> Maybe Integer
findPositive     []   = Nothing
findPositive      (x:xs)   |   x > 0       =  Just x
                           |   otherwise  =  findPositive xs
```



Типтер класы

Типтер класы кейінірек егжей тегжей қарастырылады. Қазір біз бұл жерде тек негізгі түсініктерін ғана береміз, себебі олар қолданушы типтерімен жұмысты жеңілдетеді.

Типтер класы бірқатар ортақ қасиеттері бар көптеген типтер жинағынан тұрады.

Мысалы, Eq типтер класына нысандары үшін теңдік класы анықталған болуы керек, яғни егер x және y айнымалылары бірдей типке жататын болған жағдайда Eq класына x == уих /= y өрнегін есептей аламыз. Барлық қарапайым типтер сонымен бірге кортеждер мен тізімдер де осы класқа кіреді, алайда теңдеулік қатынастар үшін анықталмаған және функция типтері Eq класына жатпайды.

Сонымен бірге Show класы да маңызды болып табылады. Show класына экранға көрсетуге мүмкін болатын нысандарды жолдарға айналдыра алатын типтер кіреді. Қарамайым типтер, кортеждер және тізімдер бұл класқа кіреді, сондықтан интерпретатор оларды енгізе алады, мысалы жолға. Функциялар бұл класқа кірмейді. Қолданушы типтер үнсіз келісім бойынша ешқандай класқа кірмейді, сондықтан бұл типтерді салыстыра алмайсын және интерпретатор да оларды басып шығара алмайды. Бұл әрине ыңғайсыз. Сондықтан типтерді анықтау кезінде оларды өзіңіз қалаған класқа жатқыза аласыз. Бұл үшін типті анықтағаннан кейін deriving кілттік сөзін қосып, жақшаларда класстарды санап шығу керек.

Мысал

```
-- Тәулік уақытын көрсететін тип
data      DayTime  =  Morning
|   Afernoon
|   Evening
|   Night dervining        (Eq,   Show)
```

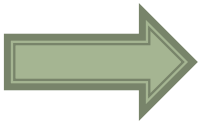
Типтерді анықтау кезінде оларды Eq және Show класына жатқызыңыз. Бұл сіздің жұмысыңызды жеңілдетеді.

Типтер синонимдері

Тілде типтердің синонимдарын анықтау мүмкіндігі бар, яғни жиі қолданылатын тип атауларын. Олар кілттік сөздер арқылы іске асады. Міне бірнеше мысал

```
type String  =  [Char]
type Person = (Name   Address)
type Name = String
type Address =     Maybe String
```

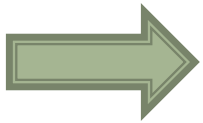
Типтер синонимдері жаңа типтерін анықтамай жай бар типтерге жаңа атаулар береді. Мысалы, Person – Name типі (String, Maybe String) -> String типіне толық сәйкес келеді. Алайда оларды қолданады, себебі біріншіден олар типтерге қысқа тау беруге көмектеседі, екіншіден кодтың түсіну деңгейін көтереді.



- Мысалы

Дүкеннің мәліметтер базасының фрагменті: торт пен жемістер бар және торттар үшін жеткізушілер көрсетіледі. Сонымен бірге, тауардың бағасы белгілі.

```
type Cake =String
type Fruit = String
type  Cost = Integer
 type Supplier = String
data Good = F Fruit  Cost
                  |    C Cake Supplier Cost
Dervining  (Eq, Show)
```



- Мысалы

Тауардың атауы бойынша бағасын алу
getCost::Good>Cost
getCost (F_ cost) = cost
getCost _ _ cost) = cost
--Мәліметтер базасының тапсырмасы
goods::[Good]
goods=[F "Apple" 12,
C "Nochka" "Konditer" 30,
C "Sever" "Oktjabr’" 23,
F "Pear" 10
C "Nochka" "Super" 34]
-- getCost функциясының орындалу нәтижесі
Main> getCost (F "Pear" 10)
10 :: Integer

- Мысал. Торттың тапсырылған атауының барлық жеткізушісінің тізімі

```
getSupplier::Cake    ->  [Good]   ->   [Supplier]
getSupplier cake     []     []
getSupplier cake    (C cake1    supplier_: goods)
                  |  cake==cake1   = supplier   :
  getSupplier cake goods
    |  otherwise    getSupplier cake goods
getSupplier cake(F    _ _:goods)
      getSupplier cake goods
-- getSupplier функциясының орындалу нәтижесі
 Main> getSuppler "Nochka" goods
["Konditer","Supper"] :: [Supplier]
```