### Haskell в питаннях (Інформатика).

**Списки**

Список – впорядкована колекція елементів одного типу

il :: [Int]

il = [3, 1, 4, 1, 200]

cl1, cl2 :: [Char]

cl1 = [’a’, ’b’, ’c’]

Конструювання списків

-- a - змінна типу (параметричний поліморфізм)

-- конструктори списків

[] :: [a] -- порожній список

(:) :: a -> [a] -> [a] -- додає елемент в голову списку

-- селектори

head :: [a] -> a -- вибирає перший елемент списку

tail :: [a] -> [a] -- вибирає «хвіст» списку

Використання «синтаксичного цукру»

cl2 = ’a’ : (’b’ : (’c’ : []))

Конструювання списків

[] :: [Int] -- порожній список

(:) :: Int -> [Int] -> [Int] -- додає елемент в голову списку

-- функції над списками

null :: [Int] -> Bool -- перевірка порожнього списку

head :: [Int] -> Int -- вибирає перший елемент списку

tail :: [Int] -> [Int] -- вибирає «хвіст» списку

* Використання «синтаксичного цукру»

il = 3 : (1 : (10 : []))

**Функції над списками (map, filter)**

map :: (a -> b) -> [a] -> [b]

map f xs = if null xs then [] else

(f (head xs)) : (map f (tail xs))

add5 :: Int -> Int

add5 x = x+5

-- map add5 [ 6,-3,0,2] = [11,2,5,7]

filter :: (a -> Bool) -> [a] -> [a]

filter p xs = if null xs then [] else

if p (head xs) then (head xs) : filter p (tail xs)

else filter p (tail xs)

divide3 :: Int -> Bool

divide3 x = (mod x 3) == 0

-- divide3 5 = False

-- divide3 6 = True

-- filter divide3 [0, 2, 3, 7, 8, 9, 11] = [0,3,9]

Відомі функції

map :: (a -> b) -> [a] -> [b]

map f xs = [f x | x <- xs]

filter :: (a -> Bool) -> [a] -> [b]

filter p xs = [x | x <- xs, p x]

Функція, що конкатенує список списків

concat :: [[a]] -> [a]

concat xss = [x | xs <- xss, x <- xs]

concat [[1,2,3], [4,5], [6]] ==> [1, 2, 3, 4, 5, 6]

З`єднання двох списків

addM :: [a] -> [a] -> [a]

addM = undefined

addM [5, 8, 91] [23, 78] = [5, 8, 91, 23, 78]

Конкатенація списку списків

concatM :: [[a]] -> [a]

concatM = undefined

concatM [[3, 4, 5], [1, 2], [10, 11, 12]] = [3,4,5,1,2,10,11,12]

Обернення списку

reverseM :: [a] -> [a]

reverseM = undefined

reverseM [78, 34, 2, 51] = [51, 2, 34, 78]

З`єднання двох списків

addM :: [a] -> [a] -> [a]

addM xs ys = if null xs then ys

else head xs : addM (tail xs) ys

Конкатенація списку списків

concatM :: [[a]] -> [a]

concatM xss = if null xss then []

else addM (head xss) (concatM (tail xss))

Обернення списку

reverseM :: [a] -> [a]

reverseM xs = if null xs then []

else addM (reverse (tail xs)) [head xs]

**Арифметичні послідовності**

Арифметичні послідовності - спосіб побудови списків (нотація ..)

l1, l2, l3 :: [Int]

l1 = [1 .. 100]

l2 = [1, 3 .. 99] -- l2 = [1, 3 .. 100]

l3 = [10, 9 .. 1]

Різниця послідовності – перший – другий елементи

Обмеження послідовності – останній елемент

Допустимі варіанти

[e1, e2 .. e3]

[e1 .. e3]

[e1, e2 ..] -- нескінченний список

[e1 ..] -- нескінченний список

**Формувачі списків**

Конструктори списків, що включають відображення, фільтр та генератор «х <- джерело», де джерело – це вираз, що задає деякий список

l4 = [x\*x | x <- [1 .. 10]] -- l4 = [1, 4, 9, .., 100]

l5 = [x\*x | x <- [1 .. 10], even x] -- l5 = [4, 16, .., 100]

Формувач може включати декілька генераторів

l6 = [x+y | x <- [1..3], y <- [10,12]] -- l6 = [11,13,12,14,13,15]

l7 = [x+y | y <- [10,12], x <- [1..3]] -- l7 = [11,12,13,13,14,15]

Формувач в загальному вигляді [exp | q1, …, qn]

qi – генератор: pat <- expr1

qi – предикат, охоронний вираз: expr2

qi - локальні імена: let n = expr3

l8 = [v | x <- [1..3], y <- [10,12], let v = xx + yy]

==> [101, 145, 104, 148, 109, 153]

Якщо генераторів декілька, то наступні генератори можуть залежати від змінних, котрі вводяться в генераторах, що розташовані раніше

l9 = [x+y | x <- [1..3], y <- [x..3]] -- l9 = [2,3,4,4,5,6]

**Нескінченні списки**

Нескінченні списки можна формувати використовуючи рекурсію, арифметичні послідовності та формувачі списків

ones :: [Int]

ones = 1 : ones -- рекурсія

-- ones = [1,1 ..] -- арифметична послідовність

numbersFrom :: Int -> [Int]

numbersFrom n = [n, n+1 ..] -- numbersFrom n = n : numbersFrom (n+1)

Нескінченний список простих чисел

primes :: [Int]

primes = 2 : [x | x<- [3,5..], ([y | y <- [1..x], mod x y == 0] ==[1,x])]

**Функції**

Визначення і використання функції

add :: Int -> Int -> Int

add x y = x + y -- add 4 7 = 11

add5 :: Int -> Int

add5 = add 5

Списки, рекурсія та if-then-else

length :: [a] -> Int

length xs = ***if*** null xs ***then*** 0 ***else*** 1 + length (tail xs)

Визначає довжину списку

-- length [2, 4, 6] = 3

sum :: Num a => [a] -> a

sum xs = ***if*** null xs ***then*** 0 ***else*** head xs + sum (tail xs)

Визначає суму всіх елементів списку

-- sum [2, 4, 6] = 12

**Визначення функції і виконання функції**

Найпростіше визначення функції в Haskell

sumList :: [Int] -> Int -- тип функції

sumList xs = expr -- означення функції: xs – параметр, expr - вираз

В виразі, що визначає функцію, можна використовувати:

Цілі і логічні операції: +, -, \*, /, mod, div, ^, ||, &&, not

Базові функції зі списками: null, head, tail

Базові конструктори списків: [], :

Умовний вираз: if e1 then e2 else e3

Рекурсію

Функція, що знаходить суму елементів списку

sumList xs = if null xs then 0 else haid xs + sumList (tail xs)

sumList [5, 12, 8] = 25

name pat11 … pat1n = expr1

…………………………………………..

name patm1 … patmn = exprm

name – ім»я , m>0, n≥0

pat11, …, patmn - зразки

expr1, …, exprm - вирази

Найпростіша форма

лише 1 рівняння (клоуз), всі зразки – імена

simple :: Int -> Int -> Int -> Int

simple x y z = x+y+z

При виклику функції

Кожному зразку-параметру відповідає вираз-аргумент

Виконується співставлення зі зразком

Обчислюється перше рівняння (клоуз), у якого співставляються всі зразки

-- останній елемент списку

last :: [a] -> a

last [x] = x

last (x:xs) = last xs

Виконання виклику last [1,2]

[1,2]=(1:[2]) співставляється з (x:xs) =>

x=1, xs=[2] і виконується виклик last [2]

[2] співставляється з [x] =>

x=2 результат виконання 2

Виконання виклику last []

Програмна помилка – немає співставлення!

Необхідно додати рівняння (клоуз)

last [] = error ”Empty list !”

**Анонімні функції**

Анонімна функція (функція без імені) створюється за допомогою λ-абстракції

\pat1 … patn -> exp (n ≥ 1)

pat1 .. patn – зразки

exp - вираз

Функцію simple можна визначити використовуючи анонімну функцію

simple = \ x y z -> x+y+z

Часто задають аргументи для функції map

add1 :: [Int] -> [Int] -- додає 1 до всіх елементів

add1 xs = map (\ x -> x+1) xs

Еквівалентно (η-редукція)

add1 = map (\x -> x+1)

**Умови (охоронні вирази)**

Умови або охоронні вирази (аналог if-then-else) використовуються, щоб робити вибір в функціях.

Загальний вигляд в одному рівнянні (клоузі) (n≥0,m≥1)

name pat1 … patn

| guard1 = expr1

……………

| guardm = exprm

Часто остання умова otherwise - функція-константа завжди == True

Спочатку виконується співставлення зі зразком щоб вибрати рівняння(клоуз) для обчислення

Перебираються послідовно умови (охоронні вирази), знаходячи першу зі значенням True

Якщо жоден з умов (охоронних виразів) не задовольняє

Виконується співставлення зі зразком для наступних рівнянь (клоузів)

Якщо немає більше рівнянь (клоузів) ==> зупинка обчислень

max :: Int -> Int -> Int

max x y | x > y = x

| otherwise = y

compBeg :: [Int] -> Char

compBeg (x : (y : \_)) | x > y = ’G’

| x < y = ’L’

compBeg \_ = ’N’

keepOnlyPos :: [Int] -> [Int]

keepOnlyPos [] = []

keepOnlyPos (x:xs) | x>0 = x : keepOnlyPos xs

|otherwise = keepOnlyPos xs

------------------------------------------

filter :: (a -> Bool) -> [a] -> [a]

filter \_ [] = []

filter p (x:xs) | p x = x : filter p xs

|otherwise = filter p xs

**Конструкції let, where, case**

sum2, sum3 :: [Int] -> Int -- sm - рекурсія і акумулятор

sum2 ys =

**let** sm :: [Int] -> Int -> Int

sm [] tot = tot

sm (x:xs) tot = sm xs (tot+x)

**in** sm ys 0

**let** – це вираз, в середині якого вводиться локальна функція sm

Область її дії – від **let** до кінця виразу після **in**

sum3 ys = sm ys 0

**where** sm :: [Int] -> Int -> Int

sm [] tot = tot

sm (x:xs) tot = sm xs (tot + x)

**where** – це частина рівняння, що визначає функцію sm

Область її дії – тіло рівняння, в якому визначається **where**

**case** – дозволяє виконати декомпозицію (співставлення зі зразком) в виразі

last :: [a] -> a

last ls = **case** ls **of**

[x] -> x

(\_:xs) -> last xs

[] -> error ”Empty list”

Еквівалент з рівняннями (клоузами0

last [x] = x

last (\_:xs) = last x

last [] = error ”Empty list”

**Двовимірний синтаксис**

В Haskell після службових слів **let, where, of, do** {} можуть обмежувати область, в якій ; закінчує вираз, як в C (Java). Але частіше використовується двовимірний синтаксис.

Вирази що входять в одну конструкцію повинні починатися з нового рядка і з одної позиції в колонці. Позиція – перший символ після службового слова**let, where, of, do**

**let** {y = a\*b; f x = (x+y)/y } **in** f c + f d

еквівалентно

**let** y = a\*b

f x = (x+y)/y

**in** f c + f d

**Функції згортки і прогонки**

foldl :: (b -> a -> b) -> b -> [a] -> b

foldl \_ z [] = z

foldl f z (x:xs) = foldl f (f z x) xs

Неформальний запис

foldl f z [x1,…,xn] = (…((z `f` x1) `f` x2) …) `f` xn

Оператор – ідентифікатор можна використовувати в інфіксній формі

f z x1 еквівалентно z `f` x1

sum xs = foldl (+) 0 xs -- sum = foldl (+) 0

length xs = foldl (\x \_ -> x+1) 0 xs

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr \_ z [] = z

foldr f z (x:xs) = f x (foldr f z xs)

Неформальний запис

foldr f z [x1,…,xn] = x1 `f` (x2 `f` ( … (xn `f` z) …))

Прогонка:

scanl :: (b -> a -> b) -> b -> [a] -> [b]

scanl \_ z [] = z

scanl f z (x:xs) = z : scanl f (f z x) xs

Неформальний запис :

scanl f z [x1,…,xn] = [z, z `f` x1, (z `f` x1) `f` x2, …]

Перший елемент прогонки – початкове значення z

Останній елемент – результат лівосторонньої згортки

last (scanl f z xs) == foldl f z xs

f z x1 еквівалентно z `f` x1

scanl (+) 0 [1,2,2,4] = [0,1,3,5,9]

scanr :: (a -> b -> b) -> b -> [a] -> [b]

Неформальний запис : foldr f z [x1,…,xn] = [ … , xn `f` z, z]

Перший елемент прогонки – результат правосторонньої згортки

head (scanr f z xs) == foldr f z xs

Останній елемент прогонки – початкове значення z

**Оператори і секції**

Оператор – функція з двома аргументами зі спеціальним іменем (складається з символів і не містить букв). Оператор можна використовувати в інфіксній формі

(.) :: (b->c) -> (a->b) -> (a->c) -- оператор - композиція функцій

f.g = \ x -> f (g x)

Для операторів вживають спеціальну форму запису – СЕКЦІЯ, котра перетворює його в функцію одного аргументу

( ^) :: Int -> Int -> Int -- функція піднесення до степені 3^2 = 9

(^2), (2^) :: Int -> Int -- секції

Неформальний опис секцій, що утворюються з оператора (^)

(^2) = \x -> (x^2)

(2^) = \x -> (2^x)

Оператор ⬄ функція

elem-*функція* elem 6 [4,б,5] ⬄`elem`-*оператор* 6 ‘elem` [4,6,5]

^ -*оператор* 2^6 ⬄ (^) – *функція* (^) 2 6

**Пріоритет і асоціативність**

Оператор – функція з двома аргументами, котру використовують в інфіксній формі запису

В виразі необхідно вказувати порядок його обрахунку при наявності декількох операторів

Пріоритет / Асоціативність / Круглі дужки

* + 2+3\*4 ---> 2 +(3\*4)
  + [1,2]++[4,3]++[7] ---> [1,2]++([4,3]++[7])
  + 3-1-2 ---> (3-1)-2

Вказівка пріоритету та асоціативності

* + Пріоритет – це ціле число від 0 до 9
  + infix (infixl, infixr) - немає (ліва, права) асоціативность

З модуля Prelude

* + infixl 9 !! -- доступ до елементів списку (нумерація від 0)
  + infixr 5 ++ -- конкатенація списків
  + infix 4 `elem`, `notElem`

**Оператори застосування функції**

Оператор застосування функції – проміжок, найвищий пріоритет 10 і ліва асоціативність

* + simple 7 4 11 ---> (((simple 7) 4) 11)

Оператор $ - аплікатор функції

($) :: (a -> b) -> a -> b

f$x = f x

Знову застосування функції

Найменший пріоритет 0 і права асоціативність

sqrt 4 + 3 +9 ---> ((sqrt 4) + 3) + 9

sqrt (4+3+9) ---> sqrt ((4+3)+9)

sqrt $ 4+3+9 ---> sqrt $ ((4+3)+9)

f (g (z x)) ---> f $ g $ z x

$ -еквівалент запису відкриваючої ( а закриваючої ) в кінці виразу

map ($3) [(4+), (10\*), (^2)] ---> [7,30,9]

Приклад використання секції та списку функцій

1. **Типи даних**

**Базові типи даних**

i :: Int -- цілі представлені в машині

i = -78

n1 :: Integer -- цілі довільної довжини

n1 = 2^(2^(2^(2^2)))

d :: Double -- дійсні з плаваючою крапкою

d = 4.5387

b :: Bool -- логічні

b = True -- True і False - конструктори

c :: Char -- символи

c = ’x’

s :: String -- рядок – список символів

s = ”Hello, world”

sum0 :: Int -> Int -> Int -- функція з двома аргументами

sum0 x y = x+y

Char – елементи типу – символи Unicode

’n’, ’\n’, ’1’ – приклади констант символів

Int, Integer – елементи типу цілі числа

Float, Double – елементи типу числа з плаваючою крапкою

Числові константи: 1, 5, 100 – поліморфні

Інколи в програмі потрібно явно вказати тип числа

* + - 10::Int -- примітка типу

[Int], [Char] - списки

* + [] і (x:xs) – конструктори
  + [], [5] --> 5:[]
  + [7,10,15] --> 7:10:15:[]

**Синоніми типів (type)**

Можливість надавати імена типам

**type** Name t1 …tn = typeEx (n ≥ 0)

* + 1. Name – ім»я типу
    2. t1, …, tn – змінні типу
    3. typeEx – вираз над типами, використовує t1, …, tn

**type** String = [Char]

**type** Point a = (a,a)

Point Double еквівалентно (Double, Double)

**Створення нових типів (data)**

Основні типи – алгебраїчні типи даних, при їх визначенні, вводяться:

* + Ім”я типу
  + Конструктори для створення значень цього типу

**data** Name t1 … tn =

Const1 t11 … t1m | … | Constp tp1 … tpk

* + Name – ім”я типу
  + t1, …, tn – змінні типу (n ≥ 0)
  + Const1, …, Constp – конструктори типу – функції, результат яких значення типу Name.
  + t11,…,t1m, …, tp1,…,tpk – поля (компоненти)

**Співставлення зі зразком, види зразків**

[] - порожній список

[1,2] - список [1,2]

pat1: pat2 - непорожній список, у якого

«Голова» співставляється з pat1

«Хвіст» співставляється з pat2

[pat1, …, patn] – список n елементів, кожний з яких співставлється з відповідним зразком pat1, …, patn

name і \_ співставляється завжди

last (\_:xs) = last xs -- одне з рівнянь означення функції last

name@pat – співставляється зі зразком pat

pat - виконується зв»язування всіх змінних з pat

name – (додатково) зв»язується з усім аргументом

headDup :: [a] -> [a]

headDup [] = []

headDup l@(x:\_) = x:l

length :: [a] -> Int

length [] = 0

length (\_:xs) = 1+ length xs

sum :: [Int] -> Int

sum [] = 0

sum (x:xs) = x + sum xs

------------------------------------

map :: (a -> b) -> [a] -> [b]

map \_ [] = []

map f (x:xs) = f x : map f xs

Головне призначення співставлення зі зразком - вказати який конструктор побудував значення і з яких елементів.

І використати цю інформацію для прийняття рішення про подальшу обробку

В Haskell - це інколи єдиний можливий спосіб для прийняття рішення

Алгебраїчний тип даних може мати багато конструкторів даних і кожний конструктор може мати нуль або багато аргументів

Далі 4 конструктори з різною кількістю аргументів

**data** AlgDataType = Constr1 Type11 Type12

| Constr2 Type21

| Constr3 Type31 Type32 Type33

| Constr4

Щоб прийняти рішення foo - що робити зі значенням типа AlgDataType , можна створити набір рівнянь (клоузів) типу

foo (Constr1 a b) = ...

foo (Constr2 a) = ...

foo (Constr3 a b c) = ...

foo Constr4 = ...

В зразках також використовуються

* + Просто змінна x – співставляється з довільним значенням
  + Підкреслення \_, як і проста змінна, співставляється з довільним значенням, але не використовується
  + В зразку виду x@(pat) pat використовується для співставлення зі значенням, а ім’я x позначає ВСЕ значення
  + Зразки можуть *гніздитися*

checkShape :: Shape -> String

checkShape (Poligon ps@(\_:(\_:\_))) = (show ((length ps) - 1)) ++ " segments"

checkShape (Square x) = "Square – side " ++ (show x )

checkShape \_ = "Else shape"

Наступна граматика визначає, що може використовуватися як зразок

pat :: = \_ | var

| var @ ( pat )

| ( Constructor pat1 pat2 ... patn )

Зауважимо, що літерали типа 2 або ‘c’ можна розглядати як конструктори без аргументів.

Пошук по шаблонам дозволяє розпізнавати конструктори типу даних:

* + Зв”язувати з різними конструкторами різні варіанти коду
  + Встановлювати змінні для кожного поля типу даного

**Тип Maybe**

Тип Maybe створено для того, щоб фіксувати виникнення помилки і не переривати обчислень

Maybe додає до значення контекст можливої невдачі

data Maybe a = Nothing | Just a

Nothing :: Maybe a --- функції конструктори

Just :: a -> Maybe a --- функції конструктори

Головне призначення - обробка ситуації error без виходу з програми

a – змінна типу (поліморфізм)

first :: [a] -> Maybe a

first [] = Nothing

first (x:\_) = Just x

saveDiv :: Float -> Float -> Maybe Float

saveDiv x y = if y == 0

then Nothing else Just (x/y)

**Тип Either**

Тип Either добавляє до невдачі значення, щоб описати, що трапилося

**data** Either a b = Left a | Right b

Right b – вірна відповідь

Left a – невдача (часто тип a просто String)

Створити операцію, що реалізує ціле ділення i/j

Вірна відповідь - j точно (без залишку) ділить i.

В інших випадках – помилка (невдача).

div1 :: Int -> Int -> Int

div1 = div

div2 :: Int -> Int -> Maybe Int

div2 \_ 0 = Nothing

div2 i j | i `mod`j /= 0 = Nothing

div2 i j = Just (i `div` j)

div3 :: Int -> Int -> Either String Int

div3 \_ 0 = Left "DivideByZero"

div3 i j | i `mod`j /= 0 = Left "NotDivisible"

div3 i j = Right (i `div` j)

**Рекурсивні типи даних**

**data** Branch a = Leaf a | Fork (Branch a) (Branch a)

**deriving** Show

Leaf :: a -> Branch a

Fork :: Branch a -> Branch a -> Branch a

Для можливості виводу об’єктів типу Branch a, потрібно щоб тип належав класу Show

brc :: Branch Char

brc = Fork (Leaf ‘r’) (Fork (Leaf ‘a’) (Leaf ‘b’))

Крона гілки

fringe :: Branch a -> [a]

fringe (Leaf x) = [x]

fringe (Fork l r) = fringe l ++ fringe r

Сума всіх “листків” гілки

sumB :: Branch Int -> Int

sumB (Leaf x) = x

sumB (Fork l r) = sumB l + sumB r

**Імена полів**

-- традиційна форма

**data** Point0 = Point0 Float Float

abs0 :: Point0 -> Float

abs0 (Point0 x y) = sqrt (x\*x + y\*y)

-- імена полів – додатково

**data** Point1 = Point1 {pointx, pointy :: Float}

З»являються функції-селектори

pointx, pointy :: Point1 -> Float

abs1 :: Point1 -> Float

abs1 p = sqrt (pointx p\*pointx p + pointy p\*pointy p)

Додатково: нова форма конструктора

Point1 {pointx = 1.1, pointy = 2.0} <==> Point1 1.1 2.0

abs1 (Point1{pointx=x,pointy=y}) = sqrt (x\*x + y\*y)

**Ізоморфні типи даних (newtype)**

Створення ізоморфного типу, структура якого повторює структуру іншого типу

**data** NewInt = NewInt Int

**newtype** MyInt = MyInt Int

Новий тип MyInt має лише один конструктор MyInt з одним полем - типу Int

Типи даних і імена конструкторів завжди вживаються в різних контекстах ==> допускається співпадіння імен типу і конструктору

Всі імена конструкторів повинні бути різними для типів, що вводяться в одному модулі

Всі імена полів і функцій в одному модулі повинні бути різними

**Модулі**

**Означення модуля**

Модуль визначає сукупність об”єктів (значення, типи, класи типів і т.д.), використовуючи *імпорт* інших модулів

Ім”я модуля – ідентифікатор, у якого перша буква –велика. Кожний модуль в своєму файлі.

Ім”я модуля NameM

Ім”я файлу NameM.hs

Імена модулів можуть “вкладатися” один в одного

A.B і A.C

Одна папка A і в ній два файли B.hs і C.hs

Ці файли починаються заголовками

**module** A.B **where** …

**module** A.C **where** …

Програма в Haskell – набір модулів

Один з них – Main

Модуль Main завжди експортує функцію main

main :: IO()

**Експорт і імпорт модулів**

Модулі *експортують* деякі об”єкти

Варіанти експорту

Нічого не вказувати (лише ім”я модуля)

Експортуються всі локальні об”єкти

НЕ експортуються імпортовані модулі

В дужках вказується що експортувати

Що не вказано – не експортується

Просто ім’я типу Type – експортує лише ім’я типу, а не конструктори

Конструктори потрібно перерахувати

Type(…) - експортує ВСІ конструктори

Можна експортувати імпортований модуль

**module** My(**module** A, …) **where**

**import** A

Модуль може *імпортувати* деякі об’єкти

Для імпорту модуля A вживається

**import** A

Імпортується ВСЕ, що експортує модуль , включаючи імпортовані модулі

**import** A(name1, name2, …)

Імпортує лише перераховані об’єкти

Type - імпортує лише ім’я типу

Type(..) - імпортує тип і всі його конструктори

Type(Const1, Const2, ..) - імпортує тип і вказані конструктори

Модуль Prelude імпортується завжди в усі модулі (по замовчуванню)

*Означення модуля*

**module** TreeM (Tree(..)), fringle) **where**

**data** Tree = Leaf a | Branch (Tree a) (Tree a)

fringle :: Tree a -> [a]

fringle (Leaf x) = [x]

fringle (Branch l r) = fringle l ++ fringe r

*Використання модуля*

**module** Main **where**

**import** TreeM(Tree(..),fringle)

main :: IO()

main = print (fringle (Branch (Leaf 1)(Leaf 2)))

**module** Main **where**

Можна не вживати (по замовчуванню)

**Конфлікти імен**

Якщо в різних модулях, що імпортуються, вживаються об’єкти з одним іменем ==> виникає конфлікт імен

Вживаються кваліфіковані імена nameM.nameOb

Явно вказується на кваліфіковане використання імен

**module** Main **where**

**import** qualified TreeM as T

main = print (T.fringle (T.Leaf ’a’))

Інколи необхідно імпортувати з модуля все крім деяких об’єктів ==> тоді їх “ховають” (hiding)

**import** TreeM hiding(fringle)

**import** Prelude hiding (sum)

Закриває функцію sum

**Дії введення-виведення.**

Тип IO a – опис дій

Значення типу IO a – це опис (алгоритм) як отримати значення типа a (”річ в собі”)

Найпростіша програма

**module** Main **where**

main :: IO()

main = putStrLn “Hello World!”

Помістимо цю програму в файл main.hs

putStrLn :: String -> IO()

Що може виконати значення IO a?

В командному рядку

Скомпілювати програму і потім виконати

ghc main.hs

main.exe

runhaskell Main.hs

ghci – інтерактивна оболонка (repl)

repl = read-eval-print loop

**Дії введення-виведення**

Всі дії (оператори) введення–виведення – елементи монади IO()

Тип дії введення-виведення - IO a або IO()

Кожна дія – визначений в системі примітив або послідовна композиція інших дій

Для об’єднання дій введення- виведення: do-нотація

main :: IO ()

main = do s <- getLine

putStrLn (work s)

getLine :: IO String -- вводить рядок

Конструкція <- єдиний спосіб отримати введений дією IO рядок

work :: String -> String

work = map toUpper s

Розподіл «грязної» і чистої частини коду

getLine – «нечиста» функція: її результат різний при різних викликах

* + - ВСІ «нечисті» функції мають тип IO a або IO()

work – чиста функція

* + - toUpper :: Char -> Char
    - **import** Data.List(toUpper) -- toUpper потрібно імпортувати

Функція, що вводить рядок

getLine :: IO String

getLine = **do** c <- getChar

**if** c == ‘\n’

**then** return “”

**else** **do** s <- getLine

return (c:s)

Дія getLine – композиція інших примітивних дій

return :: a -> IO a

Дозволяє внести чисте значення типу a в середину типу IO a

Вживання **let** в середині блоку **do**

getBool :: IO Bool

getBool = **do** c<- getChar

**let** v = (c==‘T’)

return v

Конструкція <- зв’язує ім’я з результатом дії введення-виведення

Конструкція let зв’язує ім’я з чистим значенням

**Файли і робота з ними**

Файл – послідовність фрагментів даних, що поступають на вхід програми і виводяться в результаті її роботи

Зовнішні імена файлів – рядки

Структура Handle – дескриптор - зв”язує назву файла (рядок) з відповідною послідовністю даних

Робота з файлами в монаді IO – модуль System.IO

Файл відкривається в певному режимі

**type** FilePath = String

**data** Handle = …..

**data** IOMode = ReadMode | WriteMode | AppendMode

| ReadWriteMode

-- стандартні файли

stdin, stdout, stderr :: Hangle

В модулі Prelude визначаються прості функції роботи з стандартними файлами stdin, stdout, stderr

putChar :: Char -> IO()

putStr :: String –> IO()

putStrLn :: String -> IO()

getChar :: IO Char

getLine :: IO String

getContents :: IO String

getContents – читає весь зміст файлу – як один рядок

Після кожного рядка файлу вставляється символ ‘\n’

openFile :: FilePath -> IOMode -> IO Handle

hClose :: Handle -> IO()

hIsEOF :: Handle -> IO Bool

hGetChar :: Handle -> IO Char

hGetLine :: Handle -> IO String

hGetContents :: Handle -> IO String

hPutChar :: Handle -> Char -> IO()

hPutStr :: Handle -> String -> IO()

hPutStrLn :: Handle -> String -> IO()

readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO()

**Програма копіювання файлів**

*Використання десктипторів файлів*

main :: IO()

main = **do** from <- getAndOpen “From: ” ReadMode

to <- getAndOpen “To: ” WriteMode

contents <- hGetContents from

hPutStr to contents

hClose to

hClose from

putStr “Done”

getAndOpen :: String -> IOMode -> IO Handle

getAndOpen prompt mode = **do** putStr prompt

name <- getLine

openFile name mode

*Використання writeFile/readFile*

main :: IO()

main = **do** f1 <- getNameFile “From: ”

f2 <- getNameFile “To: ”

s <- readFile f1

writeFile f2 s

putStr “Done”

getNameFile :: String -> IO String

getNameFile prompt = **do** putStr prompt

getLine

*Використання аргументів командного рядка*

main :: IO()

main = **do** [f1, f2] <- getArgs

s <- readFile f1

writeFile f2 s

putStr “Done”

Функція getArgs знаходиться в модулі System.Environment

**Класи типів**

*Клас типів* в Haskell визначає набір операції. Ми можемо вибрати деякі типи, як *екземпляри класу*, котрі підтримують цю множину. Тобто класу типів відповідає множина типів, котрі мають операції визначені для цього класу

**Поняття класу типів**

**class** Name a **where**

f1 :: y1 -> …

………..

fn :: yn -> ….

Name – ім’я класу

a – змінна типу, використовується в функціях f1 … fn для посилання на тип класу

Клас може мати контекст (*суперклас*)

**class** A1 a => Name a **where**

………

Клас A1 суперклас класу Name

Клас Name успадковує всі операції класу A1 і додає свої операції

Аналогії

Програмістська – інтерфейс. Інтерфейс визначає набір методів, котрі можна застосовувати до всіх типів, що підтримують даний інтерфейс

Математична – алгебраїчна система

Алгебраїчна система – це набір операцій і властивостей цих операцій

**Екземпляри класів**

**instance** Name t **where**

f1 x1 … = …

……..

fn xn … = …

t екземпляр класу Name, містить означення всіх методів

Тип t можна об’явити екземпляром класу типів Name, якщо показати як цей тип реалізує операції класу Name

Якщо тип t залежить від іншого типу a, то можливо потрібно накласти обмеження на цей тип

**instance** A1 a => Name t a **where**

…………………….

**Клас типів Eq, Ord, Enum**

Для різних типів операція рівності виконується по різному, але хочемо всюди називати її іменем ==.

Але не всі типи можна порівнювати (функції)

Операція == зв’язується з класом типів Eq.

Кожний тип, що входить в клас Eq, повинен реалізовувати операцію ==

З класом типів зв’язується множина операцій (функцій, методів)

Кожний тип може бути екземпляром декількох класів

elem :: Eq a => a -> [a] -> Bool

x `elem` [] = False

x `elem` (y:ys) = (x==y) || (x `elem` ys)

Eq a – КОНТЕКСТ (contex)- обмеження на тип

Може бути декілька контекстів

(Eq a, Enum a) => [a] -> a

(Ord a, Num b) => a -> b -> b

*Типи в яких визначена операція рівність*

**class** Eq a **where**

(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool

x==y = not (x/=y)

x/=y = not (x==y)

Мінімальне визначення класу == або /=

*Приклади екземплярів класу Eq*

**instance** Eq a => Eq [a] **where**

[] ==[] = True

(x:xs) == (y:ys) = x==y && xs == ys

\_ ==\_ = False

**data** Maybe a = Nothing | Just a

**instance** Eq a => Eq (Maybe a) **where**

Just x == Just y = x==y

Nothing ==Nothing = True

\_ == \_ = False

*Повністю впорядковані типи даних*

**data** Ordering = LT | EQ | GT

**class** Eq a => Ord a **where**

compare :: a -> a -> Ordering

(<),(<=),(>=),(>) :: a -> a-> Bool

min, max :: a -> a-> a

compare x y = **if** x==y **then** EQ **else** **if** x<=y **then** LT **else** GT

x<=y = compare x y /= GT

x<y = compare x y == LT

x>=y = compare x y /= LT

x>y = compare x y == GT

max x y = if x<=y then y else x

min x y = if x<=y then x else y

Досить визначити compare або <=

Визначає операції над повністю впорядкованими типами (перелічувані)

succ і pred – наступний і попередній елемент заданого значення

fromEnum і toEnum - перетворюють Enum в Int і Int в Enum

enumFrom, enumFromThen, enumFromTo, enumFromThenTo – формують арифметичні послідовності

[1,3..] *еквівалентно* enumFromThen 1 3

Екземпляри цього класу

Перелічувані типи, котрі створюються означеннями виду:

**data** Name1 | … | Namek

Name1, …, Namek – конструктори константи

Bool, Ordering

Char на основі функції Char -> Int

Числові типи - Int, Integer, Float, Double

**Автоматичне визначення екземплярів класів типів**

При об’яві **data** t нового типу t, можна автоматично визначати його екземпляром класу Eq, Ord, Enum, Bounded, Show або Read

**data** BinTree a = Empty

| Node a (BinTree a) (BinTree a)

**deriving** (Show, Read, Eq, Ord)

Представлення бінарного дерева у виді рядка формується використовуючи конструктори аналогічно побудові відповідних констант

Node 7 (Node 2 Empty (Node 4 Empty Empty))

(Node 8 Empty Empty)

Бінарне дерево з коренем 7 і лівим сином (вузли 2 і 4) та правим сином (вузол 8)

Якщо необхідно показувати по іншому данні типу BinTree, то потрібно

Не включати в **deriving** Show

Зробити тип BinTree екземпляром класу Show

**instance** (Show a) => Show (BinTree a) **where**

show Empty = “.”

show (Node x l r) = “(“++ show x ++ show l ++ show r ++ “)”

(7(2.(4..))(8..))

**Числові класи**

Всі ці класи визначаються в Prelude

В окремих бібліотеках визначаються типи :

* Ratio (Data.Ratio) – раціональний тип (дріб)
* Complex (Data.Complex) – комплексний тип
* Num –батьківський клас для всіх числових класів типів
  + Num - підклас Eq, але не Ord
    - Немає порядку на комплексних числах
  + Num підтримує операції і функції
    - (+),(-),(\*) :: a -> a -> a
    - negate,abs, signum :: a -> a
    - fromIntegral :: Integer -> a
* Num не підтримує ділення
  + Двоє видів ділення ==> два неперетинаючі класи
  + Integral – цілочисленне ділення і залишок
    - Його екземпляри – Int і Integer
  + Fractional – підтримує звичайне ділення (/)
    - Його екземпляри – Float і Double
    - Має підклас Floating – тригоном., логариф. та експотенц. функції
  + Real – перетин класів Num і Ord
  + RealFrac – підклас Real і Fractional
  + RealFloat – доступ до компонент числа з плаваюч.крапкою
    - exponent (експонента) і significand (мантиса)

**Комплексні і раціональні числа**

Комплексні числа визначаються в модулі Data.Complex

Будує тип з типів класу RealFloat

Являється екземпляром класу Floating

**data** (RealFloat a) => Complex a = !a :+ !a

deriving (Eq, Show, Read)

* + - :+ конструктор даних
    - 5 :+ 6, 8.1 :+ 12.3 – комплекні числа
    - ! - прапорець строгості (компоненти вираховуються зразу)

Раціональні числа визначаються в модулі Data.Ratio

Будує тип з типів класу Integral

Являється екземпляром класу RealFrac

(%) :: Integral a => a -> a-> Ratio a

Функція, що будує раціональне число з двох цілих

НЕ конструктор

numerator, denominator :: Integral a => Ratio a -> a

Раціональні числа – НЕ унікальні, хоч і мають канонічну форму

* + numerator (x%y) ≠ x *але* realPart (x :+ y) = x

**Вступ до монад**

Монади – контейнерні типи даних, що являються екземплярами класу Monad

Головна мета класу – ввести операції введення-виведення (мають побічні ефекти і не детерміновані) в чисту функціональну мову програмування, що детермінована

Математично: клас Monad визначає набір операцій, котрі зв’язують обчислення над даними типу, котрий є екземпляром класу, в деяку *послідовність* дій, додаючи тим самим *імперативність*

Монада (екземпляр класу Monad) – контейнерний тип даних, в якому дані зв’язуються один з одним певною стратегією обчислень

Стратегія зв’язування двох обчислень залежить від виду монади.

Кожний екземпляр класу – своє зв’язування

**Задача з архівом (БД) даних**

Архів (БД) містить дані про фізичних осіб

Інформація про фізичну особу - тип Person

Дві функції, що працюють з архівом даних: mother і father

data Person = ….

mother, father :: Person -> Maybe Person

mother p = …

father p = …

Головна задача – отримати інформацію про одного з предків

Дідусь по лінії матері – mf p

Дідусь матері по лінії батька – mff p

* mf :: Person -> Maybe Person
* mf p = **case** (mother p) **of**
* Nothing -> Nothing
* Just m -> father m
* ------------------------------------------
* mff :: Person -> Maybe Person
* mff p = **case** (mother p) **o**f
* Nothing -> Nothing
* Just m -> **case** (father m) **of**
* Nothing -> Nothing
* Just mf -> father mf

**Клас Monad**

**class** Applicative m => Monad m **where**

return :: a -> m a

(>>=) :: m a -> (a -> m b) -> m b

(>>) :: m a -> m b -> m b

fail :: String -> m a

-----мінімум (>>=) і return

m >> k = m >>= (\ \_ -> k)

fail s = error s

return – перетворює значення типа a в значення монади типа m a (“заносить” в контейнер)

(>>=) – “зв’язує” (комбінує) значення типа a, котре знаходиться в значенні монади типа m a, з функцією \v -> m b (використовує значення v типа a і повертає значення монади типа m b)

Результат “зв’язування” – значення монади типа m b

(>>) – схоже, але функція НЕ використовує значення типа a з монади типа m a, незалежно формуючи значення монади типа m b

**Нотація do**

mf, mff :: Person -> Maybe Person

mf p = **do** m <- mother p

father m

-- mf p = **do** { m <- mother p; father m}

mff p = **do** m <- mother p

mf <- father m

father mf

-- mff p = **do** { m <- mother p; mf <- father m; father mf}

* Можна зберегти імена, переписавши без **do**-нотації

mf p = mother p >>= \m ->

father m

mff p = mother p >>= \m ->

father m >>= \mf ->

father mf

* Правила переходу від **do**–нотації до звичайної
  + x <- expr1; ... ==> expr1 >>= \x -> …
  + expr2; … ==> expr2 >>= \\_ -> …

**Монада Maybe**

**instance** Monad Maybe **where**

Nothing >>= f = Nothing

(Just x) >>= f = f x

return = Just

* + η-редукція -- return x = Just x

mf, mff :: Person -> Maybe Person

mf p = (return p) >>= mother >>= father

mff p = (return p) >>= mother >>= father >>= father

* + *Аксіоми монад*

(return a) >>= k = k a -- return – ліва одиниця для >>=

m >>= return = m -- return – права одиниця для >>=

m >>= (\x -> (k x >>= h)) = (m >>= k) >>= h

**Тип State**

**newtype** State s a = State {runState :: (s -> (a,s))}

* Обчислення працює зі станом s і має результатом тип a
  + Якщо st ϵ State, то
    - runState st отримує функцію f :: s -> (a,s)
    - функції f передаємо стан s
    - отримується результат a і новий стан s’
* Функції з типом State s a
  + runState :: State s a -> s -> (a,s)
  + evalState :: State s a -> s -> a
    - evalState st s = fst (runState st s)
  + execState :: State s a -> s -> s
    - execState st s = snd (runState st s)
* Функції, що будують об”єкти типу State

get :: State s s

get = State (\s -> (s,s))

put :: s -> State s ()

put s = State (\\_ -> ((),s))

**Монада State**

Тип State являється екземпляром класу Monad

**instance** Monad (State s) **where**

return :: a -> State s a

return a = State (\s -> (a,s))

(>>=) :: (State s a) -> (a -> State s b) -> (State s b)

(State x) >>= f = State(\s -> **let** (v,s’) = x s

(State q) = f v

**in** q s’)

* + return - функція переходу стану, яка встановлює значення але не змінює стан
  + >>= створює функцію переходу стану, котра застосовує правий аргумент f до значення v і нового стану s’ від лівого аргументу (State x)
    - Можливо коротше трактування >>=
      * + State (\s -> **let** (v,s’) = x s
        + **in** runState (f v) s’)

**Приклади монади State**

**import** Control.Monad.State

greeter :: State String String

greeter = **do** name <- get

put "tintin"

return ("hello, " ++ name ++ "!")

--*runState :: State s a -> s -> (a,s)*

--*get = State (\s -> (s,s))*

*--put s = State (\\_ -> ((),s))*

*--return a = State (\s -> (a,s))*

runState greeter "Edit“

*"Edit“ ==>*

greeter = **do**

*s дія (a,s)*

“Edit” name <- get (“Edit”, “Edit”)

“Edit” put "tintin“ ((), “tintin”)

“tintin” return ("hello, " ++ name ++ "!") (“hello, Edit!”, “tintin”)

*==> (“hello, Edit!”, “tintin”)*

**import** Control.Monad.State

greeter :: State String String

greeter = **do** name <- get

put "tintin"

return ("hello, " ++ name ++ "!")

test :: State Int Int

test = **do**

put 3

modify (+1)

get

-- runState test 0

-- execState test 0

Функція modify комбінація функцій get і put

modify :: (s -> s) -> State s ()

modify f = get >>= \x -> put (f x)

Обрахунок найбільшого спільного дільника двох натуральних чисел, використовуючи “схований стан” в середині монади State – (Int,Int)

*--* ***newtype*** *State s a = State {runState :: (s -> (a,s))*}

gcdS:: State (Int,Int) Int

gcdS = **do** (x,y) <- get

**if** x == y

**then** return x

**else if** x < y

**then** **do** put (x,y-x)

gcdS

**else** **do** put (x-y,y)

gcdS

grComDiv :: Int -> Int -> Int

grComDiv x y = fst(runState gcdS (x,y))

**Монада Either**

Тип Maybe створено для того, щоб фіксувати виникнення помилки і не переривати обчислень

Maybe додає до значення контекст можливої невдачі

**data** Maybe a = Just a | Nothing

Тип Either дозволяє до невдачі значення , щоб описати, що трапилося

**data** Either a b = Left a | Right b

Right b – вірна відповідь

Left a – невдача (часто тип a просто String)

Створити операцію, що реалізує ціле ділення i/j

Вірна відповідь - j точно (без залишку) ділить i.

В інших випадках – помилка (невдача).

div1 :: Int -> Int -> Int

div1 = div

div2 :: Int -> Int -> Maybe Int

div2 \_ 0 = Nothing

div2 i j | i `mod`j /= 0 = Nothing

div2 i j = Just (i `div` j)

div3 :: Int -> Int -> Either String Int

div3 \_ 0 = Left "DivideByZero"

div3 i j | i `mod`j /= 0 = Left "NotDivisible"

div3 i j = Right (i `div` j)

Тип Either – монада

**instance** Monad (Either a) **where**

return = Right

Left l >>= \_ = Left l

Right r >>= k = k r

* Реалізувати функцію f i j k = (i / k) + (j / k)

Вірна відповідь - k точно (без залишку) ділить i i j.

Використовуючи div3 і монаду

div3 :: Int -> Int -> Either String Int

div3 \_ 0 = Left "DivideByZero"

div3 i j | i `mod`j /= 0 = Left "NotDivisible"

div3 i j = Right (i `div` j)

full :: Int -> Int -> Int -> Either String Int

full i j k = **do**

q1 <- i `div3` k

q2 <- j `div3` k

return (q1 + q2)

**Аналізатори**

**Тип для аналізатора (Parser)**

Аналізатор – функція, що обробляє вхідний рядок символів, аналізуючи його фрагменти (лексеми), щоб побудувати складну структуру даних AST (Abstract Syntax Tree).

**newtype** Parser a =Parser { parse :: String -> Maybe (a,String) }

Аналізатор – функція, що по рядку повертає результат із Maybe (a,String)

Невдача синтаксичного аналізу => Nothing

Успішний синтаксичний аналіз => Just (v,st) :

v значення типа a (розпізнане в результаті аналізу і обробки префіксу аргументу);

st – нерозпізнаний суфікс аргументу

Parser a – тип і Parser f - конструктор,

* + - f :: String -> Maybe (a,String) - аналізатор

parse – селектор, що вибирає аналізатор f

item :: Parser Char

item =

Parser (\s -> **case** s of

“” -> Nothing

(c:cs) -> Just (c,cs)

sat p – розпізнає символ, що задовольняє предикату p

sat :: (Char -> Bool) -> Parser Char

sat p = Parser $ \s ->

**case** s **of**

“” -> Nothing

(c:cs) -> **if** p c **then** Just (c,cs) **else** Nothing

char c – розпізнає на вході символ c і його повертає

char :: Char -> Parser Char

char c = sat (==c)

digit – розпізнає цифру і повертає її

digit :: Parser Char

digit = sat isDigit

oneOf s – розпізнає символ, що являється одним з s

oneOf :: [Char] -> Parser Char

oneOf s = sat (\c -> elem c s) -- sat (flip elem s)

**Parser екземпляр класу Monad**

* + 1. return :: a -> m a
    2. (>>=) :: m a -> (a -> m b) -> m b

**instance** Monad Parser **where**

return a = Parser (\s -> Just (a,s))

p >>= f = Parser (\s -> **case** parse p s **of**

Nothing -> Nothing

Just (a,s1) -> **let** Parser g = f a **in** g s1 )

При побудові аналізаторів використовують do-нотацію

**Прості аналізатори, виконання аналізатора**

spaces – розпізнає нуль або багато символів проміжку \n\t\r

spaces :: Parser ()

spaces = many (sat isSpace) >> return ()

lexem p - розпізнає конструкцію p і, можливі проміжки за нею

lexem :: Parser a -> Parser a

lexem p = **do** {a <- p; spaces ; return a}

reserved s - розпізнає рядок s і проміжки, нічого не повертає

reserved :: String -> Parser ()

reserved s = **do** { \_ <- string s; spaces}

parens p - розпізнає конструкцію p в дужках

parens :: Parser a -> Parser a

parens p *=*

**do** reserved "("

n <- lexem p

reserved ")"

return n

**Бібліотека Parsec**

Промислова бібліотека для створення синтаксичних аналізаторів

Базується на монадах

Дозволяє аналізувати контекстно-вільні граматики

* + 1. Найкращі результати при використанні LL(1)-граматик
    2. Використовує передбачаючий (predictive) аналіз

**data** ParsecT s u m a

Аналізатор з вхідним потоком s, що використовує стан u і повертає тип a

Перетворювач монад з монадою m в основі

**type** Parsec s u = [ParsecT](file:///C:\Program%20Files\Haskell%20Platform\8.0.1\lib\extralibs\doc\parsec-3.1.9\html\Text-Parsec-Prim.html) s u [Identity](file:///C:\Program%20Files\Haskell%20Platform\8.0.1\doc\html\libraries\base-4.9.0.0\Data-Functor-Identity.html)

Перетворювач монад, в основі якого лежить монада Identity

**type** Parser  = [Parsec](file:///C:\Program%20Files\Haskell%20Platform\8.0.1\lib\extralibs\doc\parsec-3.1.9\html\Text-Parsec.html) [String](file:///C:\Program%20Files\Haskell%20Platform\8.0.1\doc\html\libraries\base-4.9.0.0\Data-String.html) ()

Вхідні дані для аналізу задаються як рядок String, стан u не використовується

**import** Text.ParserCombinators.Parsec

Для використання потрібно її імпортувати

**Робота з аналізатором в бібліотеці Parsec**

space, letter, digit, anyChar, alphaNum, newline :: Parser Char

Розпізнає один символ і його повертає

* + 1. space - типа проміжок (задовольняє предикату isSpace)
    2. letter, digit – букв ‘A’ - ’Z’, ‘a’ - ’z’ , цифра ‘0’ – ‘9’
    3. anyChar – довільний символ
    4. alphaNum – буква або цифра ‘0’ – ‘9’
    5. newline - символ ‘\n’

char :: Char -> Parser Char

char c - розпізнає символ c і його повертає

string :: String -> Parser String

string str – розпізнає рядок str і його повертає

oneOf, noneOf :: String -> Parser Char

one of st (noneOf st) – розпізнає один зі символів рядка st (якого немає в st) і його повертає

spaces :: Parser ()

Пропускає 0 або більше символів isSpace

eof :: Parser ()

eof – повертає успіх лише на кінці вхідного рядка

many, many1 :: Parser a -> Parser [a]

many p (many1 p) - розпізнає 0 або більше p (розпізнає 1 або більше p), повертаючи їх список p

sepBy, sepBy1 :: Parser a -> Parser b -> Parser [a]

sepBy p sep (sepBy1 p sep) – розпізнає 0 або більше p, розділених sep (розпізнає 1 або більше p , розділених sep), повертаючи їх список p

(<|>) :: Parser a -> Parser a -> Parser a

Комбінатор детермінованого вибору

p <|> q або try p <|> q

Спочатку аналіз p, якщо успіх ==> результат

якщо невдача ==> застосовується q

Якщо для розпізнавання невдачі p, потрібно прочитати більше одного символу, то try p <|> q

parse :: Parser a -> String -> String -> Either ParserError a

parse p fl input

* + - Виконує аналізатор p на вході input
    - fl використовується в повідомленні про помилку

Може бути “”

* + - Повертає: помилку Left або успіх Right

Left ep – ep: значення типа ParserError

Right a – a: розпізнане значення

main :: IO()

main = **case** (parse numbers “” “11,2,45”) **of**

Left err -> print err

Right xs -> print (sum xs)

numbers :: Parser [Integer]

numbers = commaSep number

commaSep :: Parser a -> Parser [a]

commaSep p = p `sepBy1` (char ',‘)

number :: Parser Integer

number = **do** {cs <- many1 digit; return (read cs)}