1. Функції
   * **Визначення функції і застосування функції**

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

Приклад:

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!”

* + **Оператори і секції**

Оператор – функція з двома аргументами зі спеціальним іменем (складається з символів і не містить букв). 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 [4,6,5] <=> `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. У виразі необхідно вказувати порядок його обрахунку при наявності декількох операторів

2+3\*4 ---> 2 +(3\*4)

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

3-1-2 ---> (3-1)-2

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`
  + **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 жодна 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

Приклади:

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 case, let, where**

let – це вираз, всередині якого вводиться локальна функція sm

Область її дії – від let до кінця виразу після in

sum2, sum3 :: [Int] -> Int

sum2 ys = let sm :: [Int]-> Int-> Int -- sm – рекурсія і акумулятор

sm [] tot = tot

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

in sm ys 0

where – це частина рівняння, що визначає функцію sm

Область її дії – тіло рівняння, в якому визначається where

sum3 ys = sm ys 0

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

sm [] tot = tot

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

case – дозволяє виконати декомпозицію (співставлення зі зразком) у виразі

last :: [a] -> a

last ls = case ls of

[x] -> x

(\_:xs) -> last xs

[] -> error ”Empty list”

Еквівалент з рівняннями (клоузами)

last [x] = x

last (\_:xs) = last x

last [] = error ”Empty list”

* + **Двовимірний синтаксис**

У 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**

Лівостороння згортка:

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

foldl \_ z [] = z

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

Правостороння згортка:

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

foldr \_ z [] = z

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

Лівостороння прогонка:

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

scanl \_ z [] = z

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

Перший елемент прогонки – початкове значення z

Останній елемент – результат лівосторонньої згортки:

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

Правостороння прогонка:

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

Перший елемент прогонки – результат правосторонньої згортки:

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

Останній елемент прогонки – початкове значення z

* + **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

Приклад використання секції та списку функцій:

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

1. Типи даних
   * **Базові типи даних**  
     data Char = ‘a’|’b’|’c’ … – символи юнікоду  
     data Int, Integer = 0|1|-1|2|… – цілі числа  
     Double, Float – числа з плаваючою крапкою  
     якщо потрібно явно вказати тип 5::Int  
     Bool – в презентацій його немає але ж він теж базовий???
   * **Списки, кортежі**  
     data List a= Nil|Cons a(List a)  
     Список – послідовність однотипних даних  
     [Int] [Char] – списки (всі елементи одного типу)  
     конструктори (x:xs) і []  
      (:) Cons  
     [] Nil  
     data Pair a b = Pair a b  
     (,) – Pair  
     (Int,Double), (Int,[Char],String) – кортежі (можливо включати елементи різних типів)  
     Кортеж – добуток типу з двома полями, значення яких описуються типами-параметрами a і b
   * **Синоніми типів (type)**  
     Можна давати імена типам  
     type Name t1..tn=typeEx(n>=0)   
     Name – ім’я типу  
     t1, …, tn – змінні типу  
     typeEx – вираз над типами, використовує t1, …, tn  
     type Values = [Int]  
     type Point a = (a,a) --Point Double = (Double,Double)
   * **Створення нових типів (data)**  
     дає можливість створювати кастомні типи даних. При їх визначенні вказуємо ім’я типу та конструктори  
     data Name t1 … tn = Const1 t11 … t1m | … | Constp tp1 … tpk   
     Name – ім’я типу  
     Const1 …Constp – конструктори типу  
     t1 .. tn – змінні типу  
     t11…t1m…tp1 ...tpk - поля  
     data Value = Int | Bool  
     data Branch = Fork (Branch a) (Branch a) | Leaf a
   * **Ізоморфні типи даних (newtype)**Ізоморфні типи – нові типи, структура яких 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
  + **Співставлення зі зразком**  
    -- паттерни  
    Головне призначення співставлення зі зразком - вказати який конструктор побудував значення і з яких елементів.  
    І використати цю інформацію для прийняття рішення про подальшу обробку  
    Щоб прийняти рішення що робити зі значенням типу, можна створити набір рівнянь (клоузів, паттернів) типу

data Op = Add | Minus | Mul | Less | Equal | Index

applyOp :: Op -> Value -> Value -> Value

applyOp Add ... =

applyOp Minus ... =

applyOp Mul ... =

applyOp Less ... =

applyOp Equal ... =

applyOp Index ... =

* + **Тип Maybe і Either**

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

data Maybe a = Nothing | Just a

Головне призначення – обробка ситуації error без виходу з програми

first :: [a] -> Maybe a

first [] = Nothing

first (x:\_) = Just x

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)

div :: Int -> Int -> Either String Double

div \_ 0 = Left "Divide by zero!"

div i j = Right (i/j)

1. Модулі
   * **Означення модулів**

Модуль визначає сукупність об’єктів (значення, типи, класи типів і т.д.), використовуючи *імпорт* інших модулів. Програма в Haskell – це набір модулів.

* + **Експорт і імпорт модулів**

Модулі *експортують* деякі об’єкти.  
module Name where буде експортувати всі імена окрім чужих  
Варіанти експорту:

1. Нічого не вказувати (лише ім’я модуля)

* Експортуються всі локальні об’єкти
* НЕ експортуються імпортовані модулі

1. 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

Модуль може *імпортувати* деякі об’єкти  
Для імпорту модуля A вживається:

1. import A

* Імпортується ВСЕ, що експортує модуль,  
  включаючи імпортовані модулі

1. 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
  + **Конфлікти імен**

Якщо в різних модулях, що імпортуються, вживаються об’єкти з одним іменем, то виникає конфлікт імен. В такому випадку:

1. Вживаються кваліфіковані імена nameM.nameOb
2. Явно вказується на кваліфіковане використання імен  
   module Main where  
   import qualified BranchM as B
3. Класи типів
   * **Поняття класу типів**  
     Клас типів - певного роду інтерфейс, який зв’язується з множиною операцій (функцій або методів). Кожен тип, що входить в певний клас, повинен реалізувати всі його операції. Також клас може мати контекст (суперклас / батьківський клас). В такому разі, він успадковує всі операції суперкласу і додає свої.
   * **Екземпляри класів**Екземпляр класу - набір операцій для конкретного (або узагальненого) типу. Тип t можна об’явити екземпляром класу типів Name, якщо показати як цей тип реалізує операції класу Name.
   * **Класи типів Eq, Ord і Enum**Eq, Ord, Enum - базові класи типів у Haskell
4. Eq – Типи, в яких визначена операція рівності (==, /=). Досить визначити одну з двох операцій
5. Ord – Повністю впорядковані типи даних. Визначені операції compare, <, <=, >=, >, min, max. Досить визначити compare або <=
6. Enum – Визначає операції над повністю впорядкованими типами (перелічувані): succ, pred, fromEnum, toEnum, enumFrom, enumFromThen, enumFromTo, enumFromThenTo. Екземплярами цього типу, наприклад, є числові типи та тип Char
   * **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 (deriving)
   * **Числові класи**Num – батьківський клас для всіх числових класів типів. Є підкласом Eq, але не Ord. Num підтримує операції і функції +, -, \*, negate, abs, signum, fromIntegral. Проте, Num не підтримує ділення. У Haskell підтримується два види ділення – цілочисленне (клас Integral) і дробове (клас Fractional).
   * **Клас Monoid**Клас типів з асоціативною бінарною операцією, що має одиницю (півгрупа з нейтральним елементом).   
     Визначення за замовчанням:  
     mappend = (<>)  
     mconcat = foldr mappend mempty  
     Мінімальне визначення включає mempty. Для екземплярів цього класу повинні виконуватись закони  
     mempty <> x = x  
     x <> mempty = x  
     Для одного типу можна визначити лише один екзкмпляр класів Semigroup і Monoid
   * **Клас Functor**Клас контейнерів, що дозволяють застосувати функцію всередині структури даних, не змінюючи саму структуру даних (операція fmap або <$>)
   * **Клас Applicative**Клас типів, котрі можуть повністю проводити обчислення всередині контейнера (контексту). Містить операції pure (заключення чистого значення в контейнер), <\*> (вибір функції що знаходиться в контейнері, і застосувати її до аргументу в контейнері  
     Мінімальне визначення: pure та <\*> або liftA2
   * **Клас Foldable**Клас контейнерів, що містить операцію згортки (обчислення деякої підсумкової інформації із усіх елементів в контейнері  
     Мінімальне визначення: fold або foldMap  
     Тип [] (списки) - екземпляр класу Foldable
7. Монади
   * **Клас Monad**

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

instance Monad Maybe where

Nothing >>= \_ = Nothing

(Just x) >>= f = f x

return = Just

(>>) = (\*>)

fail \_ = Nothing

η-редукція -- return x = Just x

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
  + **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 >>= \\_ -> ...

* + **Монади Maybe і Either**

instance Functor Maybe where

fmap \_ Nothing = Nothing

fmap f (Just a) = Just (f a)

instance Applicative Maybe where

pure x = Just x

Just f <\*> m = fmap f m

Nothing <\*> \_m = Nothing

liftA2 f (Just x) (Just y) = Just (f x y)

liftA2 \_ \_ \_ = Nothing

Just \_ \*> m2 = m2

Nothing \*> \_m2 = Nothing

instance Monad Maybe where

Nothing >>= \_ = Nothing

(Just x) >>= f = f x

return = Just

(>>) = (\*>)

fail \_ = Nothing

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 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 – правильна відповідь
* Left a – невдача (часто тип a просто String)

Тип Either – аплікативний функтор і монада

instance Functor (Either a) where

fmap \_ (Left e) = Left e

fmap f (Right y) = Right (f y)

instance Applicative (Either a) where

pure = Right

Left l <\*> \_ = Left l

Right f <\*> r = fmap f r

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)

Правильна відповідь – k точно (без залишку) ділить i i j.

Використовуючи div3 і монаду

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

full i j k = do

q1 <- i `div3` k

q2 <- j `div3` k

return (q1 + q2)

* + **Монада State**

Тип State

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

Обчислення працює зі станом s і має результатом тип a

Дані типу State s a - називають “функціями зміни стану”

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’

Тип State є екземпляром класу Monad

class Applicative m => Monad m where

return :: a -> m a

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

Монада State

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

(State q) = f v

in q s’)

* return - функція зміни стану, яка встановлює значення але не змінює стан
* >>= створює функцію зміни стану, котра застосовує правий аргумент f до значення v і нового стану s’ від лівого аргументу (State g)

Можливо коротше трактування >>=

State (\s -> let (v,s’) = g s

in runState (f v) s’)

Функції з типом State s a

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

* Функція-селектор runState (State g) = g
* g :: 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))

state :: (s -> (a,s)) -> State s a

state f = State (\s -> f s)

* + **Монада Sparse**

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

newtype Sparse a = Sparse { 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

Sparse a – тип і Sparse f - конструктор,

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

parse – селектор, що вибирає аналізатор f

item :: Sparse Char

item =

Sparse (\s -> case s of

“” -> Nothing

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

Клас Functor

fmap :: (a -> b) -> f a -> f b

instance Functor Sparse where

fmap f (Sparse cs) = Sparse (\s -> case cs s of

Nothing -> Nothing

Just (a, s1) -> Just (f a, s1))

Клас Applicative

pure :: a -> f a

(<\*>) :: f(a->b) -> f a -> f b

instance Applicative Sparse where

pure a = Sparse (\s -> Just (a,s))

(Sparse cf) <\*> (Sparse ca) =

Sparse (\s -> case cf s of

Nothing -> Nothing

Just (f, s1) -> case ca s1 of

Nothing -> Nothing

Just (a, s2) -> Just (f a, s2) )

string st – розпізнає на вході рядок st і його повертає

* string :: String -> Sparse String
* string "" = pure “”
* string (c:cs) = (:) <$> (char c) <\*> (string cs)

Клас Monad

return :: a -> m a

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

instance Monad Sparse where

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

p >>= f = Sparse (\s -> case parse p s of

Nothing -> Nothing

Just (a,s1) -> let Sparse g = f a

in g s1 )

При побудові аналізаторів використовують do-нотацію.

Детермінований оператор вибору

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

p <|> q = Sparse (\s -> case parse p s of

Nothing -> parse q s

res -> res)

* + **Монада List**

instance Monad [] where

return x = [x]

xs >>= f = concatMap f xs

Розглянемо простий приклад:

add12 :: Int -> [Int]

add12 x = [x+1, x+2]

addAll12 :: [Int]

addAll12 = [10, 20, 30] >>= add12

Обчислення addAll12:

addAll12 = [10, 20, 30] >>= add12

= concatMap add12 [10, 20, 30]

= concat [add12 10, add12 20, add12 30]

= concat [ [11,12], [21, 22], [31, 32]] = [11, 12, 21, 22, 31, 32]

Простий приклад в монаді List:

instance Monad [] where

return x = [x]

xs >>= f = concat (map f xs)

Розглянемо приклад:

addList :: [Int] -> [Int] -> [Int]

addList xs1 xs2 = do n1 <- xs1

n2 <- xs2

return (n1 + n2)

* + **Функція guard**

guard t – перериває обчислення, якщо її аргумент t не True

guard :: (MonadPlus m) => Bool -> m()

guard True = return ()

guard False = mzero

Приклади виконання в різних монадах:

Тип результату: Maybe () або [()]

guard (5>2) :: Maybe () ==> Just ()

guard (5>7) :: Maybe () ==> Nothing

guard (5>2) :: [()] ==> [()]

guard (5>7) :: [()] ==> []

Тип результату: Maybe () або [()]

* guard (5>2) :: Maybe () ==> Just ()
* guard (5>7) :: Maybe () ==> Nothing
* guard (5>2) :: [()] ==> [()]
* guard (5>7) :: [()] ==> []

Результат виконання guard разом з >>

guard (5>2) >> return ‘A’ :: [Char] ==> [‘A’]

* guard спрацювала успішно
* Результат порожній кортеж – елемент списку [()]
* Операція >> його ігнорує і формує результат [‘A’]

guard (5>7) >> return ‘A’ :: [Char] ==> []

* guard не спрацювала
* Результат порожній список []
* Операція >> немає чого ігнорувати і формує результат []

Розглянемо, як виконується

do {guard test ; return 1} :: [Int]

* guard test >>= (\\_ -> return 1) :: [Int]
* concat $ map (\\_ -> [1]) (guard test)
  + Якщо test == True, то guard True = [()] і тоді
  + concat $ map (\\_ -> [1]) [()] = concat [[1]] = [1]
  + Якщо test == False, то guard False = [] і тоді
  + concat $ map (\\_ -> [1]) [] = concat [] = []

Приклади функції guard:

sevenOnly :: [Int]

sevenOnly = [1..50] >>= (\x ->

guard (‘7’ ‘elem’ show x) >> return x)

Те саме в do-нотації:

sevenOnly = do x <- [1..50]

guard (‘7’ ‘elem’ show x)

return x

full23 :: [Int]

full23 = do num <- [1..20]

guard (even num)

guard (num ‘mod’ 3 == 0)

return num

Це обчислення може трактуватися як вибір num з діапазону 1..20, а потім перевірка чи ділиться воно на 2 та 3.

* + **Формувачі списків**

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

lst1 = [x\*x | x <- [1 .. 10]]

lst2 = [x\*x | x <- [1 .. 10], even x]

Формувач може включати декілька генераторів:

lst3 = [x+y | x <- [1..3], y <- [10,12]]

lst4 = [x+y | y <- [10,12], x <- [1..3]]

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

lst5 = [x+y | x <- [1..3], y <- [x..3]]

Приклад: усі парні числа від 1 до 100:

evensUpTo100 :: [Int]

evensUpTo100 = [n | n<-[1..100], even n]

Формувач у загальному вигляді:

[exp | q1, …, qn]

* qi – генератор :: pat <- expr1
* qi – предикат, охоронний вираз :: expr2
* qi – локальні імена :: let n = expr3

У формувачі списку:

* Компоненти справа від | :: виконуються підряд – зліва направо
* Компоненти виду a <- selects :: елементи списку selects
* Компоненти без <- :: логічні вирази, якщо значення виразу False, то поточний елемент відкидається

Допустимі let - оператори :: як в do–нотації без in:

lst6 = [v | x <- [1..3], y <- [10,12], let v = x\*x + y\*y]

1. Різне
   * **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

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

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

Функція, що вводить рядок:

getLine :: IO String

getLine = do c <- getChar

if c == ‘\n’

then return “”

else do s <- getLine

return (c:s)

Функція, що дозволяє внести чисте значення типу a всередину типу IO a

return :: a -> 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 :: Handle

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

getChar :: IO Char

getLine :: IO String

getContents :: IO String

putChar :: Char -> IO()

putStr :: String –> IO()

putStrLn :: String -> IO()

print :: Show a => a -> IO()

getContents – читає весь зміст файлу – як один рядок

Після кожного рядка файлу вставляється символ ‘\n’

Основні функції роботи з файлами:

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()

Приклади програм копіювання файлів:

*Використання дескрипторів 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

1. Використання дескрипторів 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

1. 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

1. 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

* + **Бібліотека Random**

Бібліотека Random розв’язує задачу генерації псевдо випадкових чисел.

Бібліотека знаходиться в модулі System.Random

RandomGen – клас, що забезпечує інтерфейс до генераторів випадкових чисел

class RandomGen g where

genRange :: g -> (Int, Int)

next :: g -> (Int, g)

split :: g -> (g, g)

Його екземпляри – типи генераторів випадкових чисел

Надає методи роботи з генераторами випадкових чисел

* + **Бібліотека Parsec**

Бібліотека, яка дозволяє створювати парсери за допомогою поєднання комбінаторів. Найкраще підходить для LL(1) граматик.

Імпортується за допомогою:

import Text.ParserCombinators.Parsec

Прості приклади комбінаторів:

<|> – оператор вибору

many1 – застосовує переданий парсер 1 раз або більше

chainl1 – лівоасоціативна функція, що поєднує парсери

Приклади використання бібліотеки Parsec з практичного заняття:

-- розпізнає число

num :: Parser Int

num = do numStr <- many1 digit

return $ read numStr

-- повертає оператор f, якщо прочитаний символ x

infOp :: String -> (a -> a -> a) -> Parser (a -> a -> a)

infOp x f = do \_ <- string x

return f

-- розпізнає множення

mulop :: Parser (Int -> Int -> Int)

mulop = infOp "\*" (\*)

-- розпізнає дужки

paren :: Parser a -> Parser a

paren p = do \_ <- string "("

v <- p

\_ <- string ")"

return v

-- розпізнає додавання або віднімання

addop :: Parser (Int -> Int -> Int)

addop = infOp "+" (+) <|> infOp "-" (-)

-- розпізнає число або число в дужках

factor :: Parser Int

factor = num <|> paren num

-- розпізнає множники і доданки відповідно

term, expr :: Parser Int

term = chainl1 factor mulop

expr = chainl1 term addop

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

Нескінченний список простих чисел

primes :: [Int]

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