І. В. Шевченко, Ю. А. Кузнецова, М. О. Сьомочкін

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ
(Частина 1. Функціональне програмування)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

І. В. Шевченко, Ю. А. Кузнецова, М. О. Сьомочкін

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ
(Частина 1. Функціональне програмування)

навчальний посібник
з виконання лабораторних робіт

Харків «ХАІ» 2021

УДК 004.432.42
Ш37

рецензенти: канд. тех. наук. Т. В. Філімончук;
канд. техн. наук, В. О. Мартовицький

Шевченко, І. В.

Ш37 Функціональне та логічне програмування (Частина 1. Функціональне програмування). І. В. Шевченко, Ю. А. Кузнецова, М. О. Сьомочкін – Навч. посібник з виконання лабораторних робіт. – Харків: Нац. аерокосм. ун-т «Харк. авіац. ін-т », 2021. – 98 с.

Розглянуті основні аспекти функціонального програмування. Проаналізовано характерні особливості функціонального підходу, розглянуто різні приклади програмного коду мовами Haskell та F#. Показано, що основним елементом функціонального програмування є функція. Докладно описані переваги функціонального програмування. Зазначено область застосування функціональних мов – ситуації, коли імперативні мови неефективні та затратні. В результаті виконання лабораторних робіт з функціонального програмування робиться висновок про те, що функціональне програмування – досить складний підхід, але його ефективність і його можливості дозволяють йому залишатися актуальним і набирати популярність.

Посібник призначений для студентів усіх форм навчання галузі знань 12 «Інформаційні технології», що виконують лабораторні роботи з дисципліни «Функціональне та логічне програмування», і розробників програмного забезпечення, які застосовують функціональні мови програмування або функціональний стиль програмування у складних проектах, переважно повязаних з паралельними обчисленнями та багатопотоковістю.

Іл. 7. Табл. 1. Бібліогр.: 15 назв.

УДК 004.432.42

© Шевченко І. В., Кузнецова Ю. А., Сьомочкін М. О., 2021
Національний аерокосмічний
університет ім. М. Є. Жуковського»
«Харківський авіаційний інститут», 2021

ЗМІСТ

ВСТУП 5
ЗМІСТОВИЙ МОДУЛЬ 6
МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ........8
Лабораторна робота № 1. Основи роботи з інтерпретатором мови Haskell / F#. Основні типи. Найпростіші функції. 8
Лабораторна робота № 2. Рекурсивні функції. Списки. 9
Лабораторна робота № 3. Призначені для користувача типи даних в мові Haskell / F#. Функції вищого порядку. 12
Лабораторна робота № 4. Рекурсивні типи даних в мові Haskell / F#. Операції введення-виведення в мові Haskell / F#. 18
Розрахункова робота. «Функціональні мови програмування: застосування та приклади». 26
ПИТАННЯ, ТЕСТИ ДЛЯ КОНТРОЛЬНИХ ЗАХОДІВ 27
ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ 29
БІБЛІОГРАФІЧНИЙ СПИСОК 31
СЛОВНИК ТЕРМІНІВ З ПРИКЛАДАМИ 32
ПРИКЛАДИ ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ МОВОЮ F# 49
ДОДАТОК А. ТЕОРЕТИЧНИЙ МАТЕРІАЛ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 1 58
ДОДАТОК Б. ТЕОРЕТИЧНИЙ МАТЕРІАЛ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 2 68
ДОДАТОК В. ТЕОРЕТИЧНИЙ МАТЕРІАЛ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 3 73
ДОДАТОК Г. ТЕОРЕТИЧНИЙ МАТЕРІАЛ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 4 86

ВСТУП

Базовим елементом функціонального програмування є функція. Вони використовуються для будь-яких розрахунків, навіть найпростіших. Змінні в традиційному для програмування розумінні (значення, яке по ходу виконання програми може змінюватися) так само замінюються функціями. Змінні в парадигмі функціонального програмування – це позначення виразів, що застосовуються для скорочення запису. Значення їм присвоюється єдиний раз, далі вони вже не змінюються. У звичайному розумінні програмування такі «змінні» позначалися б як константи (const в C або final в Java).

Програма, написана імперативною мовою програмування, зберігає свій стан за допомогою змінних. Для збереження стану функціональним програмам змінні не потрібні. Стан зберігається в параметрах функції. Для збереження стану, а потім його зміни, необхідно писати рекурсивную функцію.

Через відсутність змінних в традиційному розумінні цього слова, у функцій у функціональній програмі немає побічних дій. Змінити значення, що зберігається в змінній, неможливо. Функція не змінює значення, що обчислюється іншою функцією, тому що вона обмежена областю видимості і незмінними змінними. Наприклад, вона не може змінити глобальні змінні або відкриті поля класу. Отже, функція впливає тільки на значення, яке їй повертається, а єдиний спосіб якось змінити це значення – це передача інших параметрів в функцію. Таким чином значно полегшується тестування програми. Можна виконувати тест функції, користуючись лише необхідними аргументами. Не треба запускати функції в порядку необхідності або відтворювати необхідний стан, потрібно всього лише передавати потрібні послідовності аргументів безпосередньо в функцію. В С ++ недостатньо перевірити просто значення, що повертається, так як функція здатна змінити в зовнішньому стані, наприклад, глобальну змінну. У ФП таких труднощів немає.

Величезна перевага функціональних програм полягає в їх можливості розпаралелювання обчислень. Вони відразу пристосовані для розпаралелювання, програмісту не треба піклуватися про це окремо. Частина коду в такій програмі не змінюється кількома різними потоками. Отже, можна просто додати потоки, не замислюючись про традиційні проблеми розпаралелювання, властиві традиційним мовам.

Інший «плюс» полягає в тому, що навіть якщо програміст створює додаток з одним потоком, компілятор здатний самостійно розпаралелити його для використання на декількох ядрах.

ЗМІСТОВИЙ МОДУЛЬ

Тема 1. Вступ до функціонального програмування. Історія розвитку функціонального програмування (ФП). Сучасний стан теорії ФП. Переваги та недоліки функціонального підходу. Основні властивості функціональних мов програмування. Приклади функціональних мов програмування. Типові задачі, що вирішуються методами ФП. Предмет вивчення і задачі дисципліни. Місце дисципліни в навчальному процесі.

Тема 2. Базові принципи функціональної мови програмування Haskell / F#. Поняття списку у функціональному програмуванні. Базисні операції для роботи зі списками. Списки й облікові структури. Програмна реалізація списків у функціональних мовах. Списки в мові Haskell / F#. Генератори списків і математичні послідовності. Кортежі. Функція – основний об’єкт функціонального програмування. Умови щодо іменування об’єктів у мові Haskell / F#. Опис і визначення функцій мовою Haskell / F#. Зразки й клози. Передача параметрів і повернення значень функціями. Виклики функцій. Використання λ-обчислень. Функція як об’єкт для передачі в інші функції. Структури й типи даних. Синоніми типів. Типи функцій у функціональних мовах. Поліморфні типи. Часткове застосування. Ледачі (відкладені) обчислення мовою Haskell / F#. Охоронні вирази і конструкції. Розгалуження алгоритму. Двовимірний синтаксис. Локальні змінні для оптимізації коду функціональною мовою F# й мовою Haskell. Використання накопичуючого параметра (акумулятора) для оптимізації процесу обчислень. Головна й хвостова рекурсії. Модулі й абстрактні типи даних. Модулі як способи структуризації й організації програм мовою Haskell / F#. Імпорт і експорт даних за допомогою модулів. Приховування даних. Абстрактні типи даних й інтерфейси.

Тема 3. Класи та їх екземпляри у мові Haskell / F#. Симбіоз парадигм функціонального й об’єктно-орієнтованого програмування. Підтримка мовою Haskell / F# об’єктно-орієнтованих механізмів й методів. Параметричний поліморфізм даних. Поняття класу і його реалізації в мові Haskell / F#. Приклади параметричного поліморфізму в імперативних і функціональних мовах, а також у мові Haskell / F#. Класи в мові Haskell / F# як спосіб абстракції дійсності. Розширений опис поняття класу в мові Haskell / F#. Методи класу – шаблони функцій для реалізації обробки даних. Мінімальний опис методів класу й зв’язок методів. Визначення класів. Спадкування й реалізація. Спадкування класів і спадкування методів. Екземпляри класів – реалізація інтерфейсів, надаваних

реалізованим класом. Реалізація методів для обробки даних. Клас – шаблон типу, реалізація класу – тип даних. Реалізація для існуючих типів. Види типів. Стандартні класи мови Haskell або мовою F#. Короткий опис всіх стандартних класів, розроблених для полегшення програмування мовою Haskell або мовою F#. Дерево спадкування стандартних класів. Типові способи використання стандартних класів мови Haskell / F#. Реалізація стандартних класів – типи в мові Haskell / F#. Порівняння з іншими мовами програмування. Порівняння понять «клас» і «реалізація класу» у мові Haskell або F# з об'єктно-орієнтованими мовами програмування (на прикладі мов C++ і Java, а також деяких інших мов). Глобальні відмінності поняття «клас» у функціональних і об'єктно-орієнтованих мовах.

Тема 4. Комбінаторна логіка й λ-обчислення. Комбінатори й обчислення за допомогою комбінаторів. Базиси в комбінаторній логіці. Використання базисних комбінаторів для вираження будь-яких обчислювальних процесів. Числа й інші математичні об'єкти у вигляді комбінаторів. Базові комбінатори. Абстракція функцій як обчислювальних процесів. Функція – об'єкт математичного дослідження. Обчислювальний процес – функція. Опис функцій як λ-виражень. Вільні та зв'язані ідентифікатори. Застосування (аплікація) значень до λ-виражень. Редукція. Теза Черча-Тьюринга. λ-обчислення як теоретична основа функціонального програмування. Інтенсіонал і екстенсіонал функцій. Правила виводу. Відповідності між обчисленнями функціональних програм і редукцією λ-виражень. Кодування даних в λ-обчисленні. Механізм кодування даних в λ-обчисленні. Редукція й обчислення у функціональних мовах. Поняття редукції. Часткові обчислення з погляду редукції λ-виражень. Стратегія редукції й стратегія обчислень. Ледача редукція.

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота № 1. Основи роботи з інтерпретатором мови Haskell / F#. Основні типи. Найпростіші функції.

Мета лабораторної роботи: набути навичок роботи з інтерпретатором мови Haskell / F#. Отримати уявлення про основні типи мови Haskell / F#. Навчитися визначати найпростіші функції.

Порядок виконання роботи:

Прочитати теоретичні відомості до лабораторної роботи (Додаток А).

Лабораторна робота складається з двох завдань, номери яких відповідають номеру варіанта. Номер варіанта дорівнює порядковому номеру студента в списку. Для того, щоб здати лабораторну роботу, кожному студенту необхідно:

1. Отримати завдання у відповідності до варіанту;
2. Виконати його (самостійно);
3. Продемонструвати програму викладачеві;
5. Оформити звіт, перевірити його у викладача, захистити лабораторну роботу.
6. Зберегти файл, який містить звіт з лабораторної роботи, та файл-архів проекту в системі Moodle (сайт mentor.khai.edu).

Зміст звіту:

1. Номер варіанта;
2. Текст завдань;
3. Текст програми;
4. При необхідності – пояснення до реалізації;
5. Результати тестів (скріншоти);
6. Висновки.

Завдання до лабораторної роботи

1. Наведіть приклад нетривіальних виразів, що належать до наступного типу:

1) [(Double, Bool, (String, Integer))]
2) [[[(Integer, Bool)]]]
3) (((Char, Char), Char), [String])
4) ([Integer], [Double], [(Bool, Char)])
5) (([Double], [Bool]), [Integer])
6) [(Integer, (Integer, [Bool]))]
7) [([Bool], [Double])]
8) [([Integer], [Char])]
9) ((Char, Integer), String, [Double])

10) (Bool, ([Bool], [Integer]))

Вимога нетривіальністю в даному випадку означає, що списки, які зустрічаються у виразах, повинні містити більше одного елемента.

2. Визначте наступні функції:

1. Функція isParallel, що повертає True, якщо два відрізки, кінці яких задаються в аргументах функції, паралельні (або лежать на одній прямій). Наприклад, значення виразу isParallel (1,1) (2,2) (2,0) (4, 2) має дорівнювати True, оскільки відрізки (1,1) - (2, 2) і (2, 0) - (4, 2) є паралельними.

2. Функція max3, за трьома цілим повертає найбільше з них.

3. Функція bothTrue:: Bool -> Bool -> Bool, яка повертає True тоді і тільки тоді, коли обидва її аргументи дорівнюватимуть True. Не використовуйте при визначенні функції стандартні логічні операції (&&, | | і т.п.).

4. Функція isRectangular, яка приймає в якості параметрів координати трьох точок на площині, і повертає True, якщо утворений ними трикутник – прямокутний.

5. Функція sort2, по двом цілим повертає пару, в якій найменше з них стоїть на першому місці, а найбільше – на другому.

6. Функція solve2 :: Double> Double> (Bool, Double), яка за двома числами, що представляють собою коефіцієнти лінійного рівняння ax + b = 0, повертає пару, перший елемент якої дорівнює True, якщо рішення існує і False в іншому випадку; при цьому другий елемент дорівнює або значенням кореня, або 0.0.

7. Функція is Triangle, яка визначає, чи можна з відрізків із заданими довжинами x, y і z побудувати трикутник.

8. Функція isSorted, що приймає на вхід три числа і повертає True, якщо вони впорядковані за зростанням або за спаданням.

9. Функція isIncluded, аргументами якої є параметри двох кіл на площині (координати центрів та радіуси); функція повертає True, якщо друга окружність цілком міститься всередині першої.

10. Функція min3, за трьома цілим повертає найменше із них.

Лабораторна робота № 2. Рекурсивні функції. Списки.

Мета лабораторної роботи: навчитися визначати рекурсивні функції. Отримати уявлення про механізм зіставлення зі зразком. Набути навичок визначення функцій для обробки списків (Haskell / F#).

Порядок виконання роботи:

Прочитати теоретичні відомості до лабораторної роботи (Додаток Б).

Лабораторна робота складається з трьох завдань, номери яких відповідають номеру варіанта. Номер варіанта дорівнює порядковому

номеру студента в списку. Для того, щоб здати лабораторну роботу, кожному студенту необхідно:

1. Отримати завдання у відповідності до варіанту;
2. Виконати його (самостійно);
3. Продемонструвати програму викладачеві;
5. Оформити звіт, перевірити його у викладача, захистити лабораторну роботу.
6. Зберегти файл, який містить звіт з лабораторної роботи, та файл-архів проекту в системі Moodle (сайт mentor.khai.edu).

Зміст звіту:

1. Номер варіанта;
2. Текст завдань;
3. Текст програми;
4. При необхідності – пояснення до реалізації;
5. Результати тестів (скріншоти).
6. Висновки.

Завдання до лабораторної роботи

1. Визначте функцію, яка приймає на вхід на вхід ціле число n і повертає список, що містить n елементів, упорядкованих за зростанням.

1) Список натуральних чисел. N = 20.
2) Список непарних натуральних чисел. N = 20.
3) Список парних натуральних чисел. N = 20.
4) Список квадратів натуральних чисел. N = 30.
5) Список кубів натуральних чисел. N = 30.
6) Список факторіалів. N = 50.
7) Список ступенів двійки. N = 25.
8) Список ступенів трійки. N = 25.
9) Список трикутних чисел. N = 50 (n-е трикутне число t_n дорівнює кількості однакових монет, з яких можна побудувати рівносторонній трикутник, на кожній стороні якого укладається n монет. Неважко переконатися, що $t_1 = 1$ і $t_n = n + t_{n-1}$).
10) Список пірамідальних чисел. N = 50 (n-е пірамідальне число p_n дорівнює кількості однакових куль, з яких можна побудувати правильну піраміду з трикутним підставою, на кожній стороні якої укладається n куль. Неважко переконатися, що $p_1 = 1$ і $p_n = t_n + p_{n-1}$).

2. Визначте наступні функції:

1) Функція GetN (L, n) виокремлення n-го елемента із заданого списку.

2) Функція OddEven (L) перестановки місцями сусідніх парних і непарних елементів в заданому списку

3) Функція ListSumm (L1, L2) складання елементів двох списків. Повертає список, складений з сум елементів списків параметрів. Врахувати, що передані списки можуть бути різної довжини.

4) Функція Reverse (L) перевертає список (перший елемент списку стає останнім, другий – передостаннім, і так далі до останнього елемента).

5) Функція removeOdd, яка видаляє із заданого списку цілих чисел всі непарні числа. Наприклад: removeOdd [1,4,5,6,10] повинен повертати [4,10].

6) Функція Set (L) повертає список, що містить поодинокі входження атомів заданого списку.

7) Функція delete:: Char> String> String, котра приймає на вхід рядок і символ і повертає рядок, в якому вилучені всі входження символу. Приклад: delete 'l' "Hello world!" Має повертати "Heo word!".

8) Функція Frequency (L) повертає список пар (символ, частота). Кожна пара визначає атом з заданого списку і частоту його входження в цей список.

9) Функція makePositive, яка змінює знак всіх негативних елементів списку чисел, наприклад: makePositive [1, 0, 5, 10, 20] дає [1, 0, 5, 10, 20].

10) Функція Insert (L, A, n) включення в список заданого атома на певну позицію.

11) Функція removeEmpty, яка видаляє порожні рядки з заданого списку рядків. Наприклад: removeEmpty ["", "Hello", "", "", "World!"] Повертає ["Hello", "World!"].

12) Функція substitute :: Char> Char> String> String, яка замінює в рядку вказаний символ на заданий. Приклад: substitute 'e' 'i' "eigenvalue" повертає "iiginvalui"

13) Функція countTrue:: [Bool]> Integer, повертає кількість елементів списку, рівних True.

14) Функція Position (L, A) повертає номер першого входження заданого атома в список.

3. Визначте функції обчислення рядів.

Номер варіанту	Рядок S
1	$S=\frac{3x^2}{4!}-\frac{5x^4}{6!}+\frac{7x^6}{8!}-\cdots+(-1)^{k-1}\frac{(2k+1)x^{2k}}{(2k+2)!}+\cdots$

Номер варіанту	Рядок S
2	$S=\frac{x}{1!}+\frac{x^2}{2!}+\dots+\frac{x^k}{k!}+\dots$
3	$S=\frac{8\sqrt{x}*x}{3!}+\frac{32\sqrt{x}*x^2}{5!}+\dots+\frac{8k^2\sqrt{x}*x^k}{(2k+1)!}+\dots$
4	$S=\frac{x}{1}+\frac{x^3}{2}+\frac{x^5}{3}+\dots+\frac{x^{2k-1}}{k}+\dots$
5	$S=\frac{2x^4(5+x)}{5!}+\frac{2x^8(9+x)}{9!}+\dots+\frac{2x^{4k}(4k+1+x)}{(4k+1)!}+\dots$
6	$S=\frac{x(4-2x+x)}{1*2}+\frac{x^3(8-4x+x)}{3*4}+\dots+\frac{x^{2k-1}(8-2kx+x)}{(2k-1)2k}+\dots$
7	$S=\frac{(2x)^2}{2!}-\frac{(2x)^4}{4!}+\frac{(2x)^6}{6!}-\dots+(-1)^{k-1}\frac{(2x)^{2k}}{(2k)!}+\dots$
8	$S=\frac{x}{3}-\frac{x^3}{5}+\frac{x^5}{7}-\dots+(-1)^{k-1}\frac{x^{2k-1}}{2k+1}+\dots$
9	$S=\frac{x^4}{3!}+\frac{x^8}{7!}+\frac{x^{12}}{11!}+\dots+\frac{x^{4k}}{(4k-1)!}+\dots$
10	$S=\frac{\ln 3}{1!}x+\frac{\ln^2 3}{2!}x^2+\dots+\frac{\ln^k 3}{k!}x^k+\dots$

Лабораторна робота № 3. Призначені для користувача типи даних в мові Haskell / F#. Функції вищого порядку.

Мета лабораторної роботи: набути навичок роботи з призначеними для користувача типами даних в мові Haskell / F#. Навчитися визначати функції вищого порядку.

Порядок виконання роботи:

Прочитати теоретичні відомості до лабораторної роботи (Додаток В).

Лабораторна робота складається з двох завдань, номери яких відповідають номеру варіанта. Номер варіанта дорівнює порядковому номеру студента в списку. Для того, щоб здати лабораторну роботу, кожному студенту необхідно:

1. Отримати завдання у відповідності до варіанту;
2. Виконати його (самостійно);
3. Продемонструвати програму викладачеві;
5. Оформити звіт, перевірити його у викладача, захистити лабораторну роботу.
6. Зберегти файл, який містить звіт з лабораторної роботи, та файл-архів проекту в системі Moodle (сайт mentor.khai.edu).

Зміст звіту:

1. Номер варіанта;
2. Текст завдань;
3. Текст програми;
4. При необхідності – пояснення до реалізації;
5. Результати тестів (скріншоти).
6. Висновки.

Завдання до лабораторної роботи

I. Визначте наступні функції з використанням функцій вищого порядку:

1) Функція countEven, що повертає кількість парних елементів в списку.

2) Функція, що обчислює скалярний добуток двох списків (використовуйте функції foldr і zipWith).

3) Функція обчислення арифметичного середнього елементів списку дійсних чисел з використанням функції foldr. Функція повинна здійснювати тільки один прохід по списку.

4) Функція quicksort, що здійснює швидку сортування списку за наступним рекурсивному алгоритму. Для того, щоб впорядкувати список xs, з нього вибирається перший елемент (позначимо його x). Решта список ділиться на дві частини: список, що складається з елементів xs, менших x і список елементів, великих x. Ці списки упорядковано (тут проявляється рекурсія, оскільки вони сортуються цим же алгоритмів), а потім з них складається результуючий список виду as ++ [x] ++ bs, де as і bs – відсортовані списки менших і більших елементів відповідно.

5) Певна в попередньому пункті функція quicksort сортує список в порядку зростання. Узагальнити її: нехай вона приймає ще один аргумент – функцію порівняння типу a -> a -> Bool і сортує список у відповідність з нею.

ІІ. Реалізуйте одне з поданих нижче завдань за допомогою функцій вищого порядку. Постарайтеся повністю виключити з визначень функцій явний прохід за списком. Номер завдання відповідає номеру варіанта студента.

1. Визначте тип, який представляє геометричні фігури на площині. Фігура може бути або окружністю (характеризується координатами центру та радіусом), прямокутником (характеризується координатами верхнього лівого і нижнього правого кутів), трикутником (координати вершин) і текстовим полем (для нього необхідно зберігати положення лівого нижнього кута, шрифт і рядок, що представляє напис) . Шрифт задається з множини трьох можливих шрифтів: Courier, Lucida і Fixedsys. Визначте наступні функції.

1) Функція **area**, що повертає площу фігури. Для текстового поля площа залежить від висоти і ширини букви в шрифті. Оскільки обрані нами шрифти – моноширинних (тобто ширина всіх букв в них однаково), Вам необхідно також визначити допоміжну функцію, для кожного шрифту повертає його габарити.

2) Функція getRectangles, зі списку фігур вибирає тільки прямокутники.

3) Функція getBound, по заданій фігурі повертає обмежуючий її прямокутник.

4) Функція getBounds, за списком фігур повертає список їх обмежують прямокутників.

5) Функція getFigure, по заданому списку фігур і координатам точки повертає першу фігуру, для якої точка потрапляє в її обмежує прямокутник. Використовуйте тип Maybe для значення, що повертається.

6) Функція move, по заданій фігурі і вектору зсуву повертає нову фігуру, зрушену щодо заданої на вказаний вектор.

2. В сучасних web-магазинах продають книги, відеокасети та компакт-диски. База даних такого магазину для кожного типу товарів повинна містити наступні характеристики:

– Книги: назва і автор;
– Відеокасети: назва;
– Компакт-диск: назва, виконавець і кількість композицій.

1) Розробіть тип даних Product, який може представляти ці види товарів.

2) Визначте функцію getTitle, що повертає назву товару.

3) На її основі визначте функцію getTitles, яка по списку товарів повертає список їх назв.

4) Визначте функцію bookAuthors, яка за списком товарів повертає список авторів книг.

5) Визначте функцію lookupTitle :: String -> [Product] -> Maybe Product, яка повертає товар з заданим назвою (зверніть увагу на тип результату функції).

6) Визначте функцію lookupTitles :: [String] -> [Product] -> [Product].

Вона приймає в якості параметрів список назв і список товарів і для кожного назви витягує з другого списку відповідні товари. Назви, яким не відповідає ніякої товар, ігнорується. При визначенні функції обов'язково потрібно скористатися функцією lookupTitle.

3. Визначте тип даних, що представляє інформацію про карту в картковій грі. Кожна карта характеризується однією з чотирьох мастей. Карта може бути або молодшою (від двійки до десятки), або картинкою (валет, дама, король, туз). Визначте функції:

1) Функція isMinor, що перевіряє, що її аргумент є молодшою картою.

2) Функція sameSuit, що перевіряє, що передані в неї карти - однієї масті.

3) Функція beats:: Card -> Card -> Bool, що перевіряє, що карта, передана їй в якості першого аргументу, б'є карту, що є другим аргументом.

4) Функція beats2, аналогічна beats, але приймаюча в якості додаткового аргументу козирну масть.

5) Функція beatsList, приймаюча в якості аргументів список карт, карту і козирну масть і повертає список тих карт з першого аргументу, які б'ють зазначену карту з урахуванням козирною масті.

6) Функція, по заданому списку карт повертає список чисел, кожне з яких є можливою сумою очок зазначених карт, розрахованих за правилами гри в «двадцять одне»: молодші карти вважаються за номіналом, валет, дама і король вважаються за 10 очок, туз може розглядатися і як 1 і як 11 очок. Функція повинна повернути всі можливі варіанти.

4. В агентстві нерухомості продають квартири, кімнати і приватні будинки. Квартира характеризується поверхом, площею і поверховістю будинку. Кімната характеризується, крім цього, площею кімнати (на додаток до площі всієї квартири). Приватний будинок характеризується тільки площею. У базі даних зберігаються пари значень, перше з яких представляє об'єкт нерухомості, а друге – його ціну. Визначте тип даних, що представляє інформацію про такі об'єкти нерухомості. Визначте наступні функції:

1) getHouses, що вибирає з бази даних тільки приватні будинки.

2) getByPrice, що вибирає з бази даних ті об'єкти нерухомості, ціна яких менше зазначеної.

3) getByLevel, що вибирає з бази даних квартири, що знаходяться на зазначеному поверсі.

4) getExceptBounds, що вибирає з бази даних квартири, які не перебувають на крайніх поверхах (перших і останніх).

Розробіть тип даних, що представляє різні вимоги до об'єктів нерухомості: бажаний тип об'єкту нерухомості, мінімальна площа, максимальна ціна, обмеження на поверх. Розробіть функцію query, яка за списком вимог вибирає з бази даних ті об'єкти нерухомості, які задовольняють всім вимогам.

5. Клавіші на клавіатурі можуть бути або керуючими, або алфавітно-цифровими. Натискання алфавітно-цифровий клавіші може супроводжуватися натисканням клавіші Shift. З керуючих клавіш нас цікавить тільки клавіша CapsLock, інші годі й розрізняти. Кожне натискання алфавітно-цифрової клавіші несе з собою інформацію у вигляді символу. Після натискання CapsLock наступні символи переводяться у верхній регістр (якщо вони не були натиснуті разом з Shift) до наступного натискання CapsLock. Якщо режим CapsLock не активований, символи, натискання з Shift, переводяться у верхній регістр. Розробіть тип даних, який представляє зазначену інформацію. Послідовність натискань клавіш представляється у вигляді списку. Основна задача полягає в тому, щоб розробити функцію, що переводять цю послідовність в рядок символів. Наприклад, послідовність натискань Shift + 'h' 'e' CapsLock 'l' 'l' Shift + 'o' CapsLock повинна дати в результаті рядок HeLLo. Визначте наступні функції:

1) getAlMum, що повертає зі списку натискань тільки натискання алфавітно-цифрових клавіш.

2) getRaw, що повертає рядок, складений з натиснутих символів без урахування інформації про Shift і CapsLock.

3) isCapsLocked, за послідовністю натискання визначає, чи залишився після неї режим CapsLock в активованому стані.

4) getString, яка переводить послідовність натискань в строку.

При реалізації функцій можна скористатися стандартними функціями toUpper і toLower, що переводять символ в верхній і нижній регістри відповідно. Вони визначені в модулі Char; щоб їх використовувати, додайте в початок програми рядок:

import Char

6. У бібліотеці зберігаються книги, газети і журнали. Книга характеризується ім'ям автора і назвою; журнал - назвою, місяцем і роком випуску; газета – назвою і датою випуску. База даних являє собою список цих об'єктів. Розробіть тип даних, який представляє об'єкти бібліотечного зберігання. Визначте наступні функції:

1) isPeriodic, яка перевіряє, що її аргумент є періодичним виданням.

2) getByTytle, що вибирає зі списку об'єктів зберігання (бази даних) об'єкти з вказаною назвою.

3) getByMonth, що вибирає з бази даних періодичні видання, випущені в зазначений місяць і вказаний рік (відзначте, що газети виходять кілька разів на місяць).

4) getByMonths, діюча так само, як і попередня, але приймає список місяців.

5) getAuthors, що повертає список авторів видань, які зберігаються в базі даних.

7. В деякій мові програмування існують такі типи даних:

Прості типи: цілі, дійсні та рядкові.

Складні типи: структури. Структура має назву і складається з декількох полів, кожне з яких, в свою чергу, має назву і простий тип.

База даних ідентифікаторів програми є список пар, що складаються з імені ідентифікатора і його типу. Розробіть тип даних, який представляє описану інформацію. Визначте наступні функції:

1) isStructured, що перевіряє, що її аргумент є складним типом.

2) getType, по заданому імені та списку ідентифікаторів (базі даних) повертає тип ідентифікатора з вказаним ім'ям (пам'ятаєте про те, що такого ідентифікатора в базі може і не виявитися).

3) getFields, по заданому імені повертає список полів ідентифікатора, якщо він має тип структури.

4) getByType, що повертає список імен ідентифікаторів вказаного типу з бази даних.

5) getByTypes, аналогічна попередній, але приймаючої замість одного типу список типів (за допомогою цієї функції можна отримати, наприклад, список всіх ідентифікаторів з числовим типом).

8. Визначимо наступний набір операцій над рядками:

– очищення: видалення всіх символів з рядка;
– вилучення: видалення всіх входжень зазначеного символу;
– заміна: заміна всіх входжень одного символу на інший;
– додавання: додавання в початок рядка зазначеного символу.

Розробіть тип даних, що характеризує операції над строками. Визначте наступні функції:

1) process, яка отримує в якості аргументу дію і рядок і повертає рядок, модифіковану у відповідність із зазначеним дією.

2) processAll, аналогічна попередній, але отримує список дій і виконує їх по порядку.

3) deleteAll, приймаюча два рядки і видаляє з другого рядка всі символи першої. При реалізації обов'язково використовуйте функцію processAll.

9. В електронній записній книжці зберігаються записи наступних видів: нагадування про дні народження знайомих, телефони знайомих і

призначені зустрічі. Нагадування складається з імені знайомого і дати (день і місяць). Запис про телефон повинна містити ім'я людини і його телефон. Інформація про призначену зустріч містить дату зустрічі (день, місяць, рік) та короткий опис (можна представляти рядком). Розробіть тип даних, що являє собою такий запис. Нотатки є списком записів. Визначте наступні функції:

1) getByName, що повертає інформацію про людину з вказаним ім'ям (його телефон і дату народження).

2) getByLetter, що повертає список людей, про яких є інформація в записнику і чиє ім'я починається на вказану букву.

3) getAssignment, що повертає по зазначеній даті список справ (інформація про призначені зустрічі і телефони друзів, котрих потрібно привітати в цей день).

10. За час навчання в семестрі студенти повинні здати певну кількість лабораторних робіт, розрахунково-графічних завдань і рефератів. Лабораторна робота характеризується назвою предмета і номером, РГЗ – назвою предмета, реферат – назвою предмета і назвою теми реферату. Розробіть тип даних, що представляє інформацію за завданням. Навчальний план студента є список, що складається з пар, перший елемент яких є завданням, а другий - номером тижні, в яку він був зданий. Якщо завдання ще не здано, другий елемент пари повинен бути порожнім (використовуйте тип Maybe). Визначте наступні функції:

1) getByTitle – повертає завдання, які необхідно здати за вказаною предмету.

2) getRef erats – повертає список тем рефератів.

3) getRest – повертає список завдань, що залишилися незданими.

4) getRestForWeek – повертає список завдань, що залишалися незданими на зазначеному тижні.

5) getPlot – створює список, що складається з пар, перший елемент яких дорівнює номеру тижні, а другий – кількістю зданих на цей тиждень завдань.

Лабораторна робота № 4. Рекурсивні типи даних в мові Haskell / F#. Операції введення-виведення в мові Haskell / F#.

<u>Мета лабораторної роботи:</u> набути навичок роботи з рекурсивними типами даних в мові Haskell / F#, а також з операціями введення-виведення у функціональних мовах програмування.

Порядок виконання роботи:

Прочитати теоретичні відомості до лабораторної роботи.

Лабораторна робота складається з двох частин, номери яких відповідають номеру варіанта. Номер варіанта дорівнює порядковому

номеру студента в списку. Для того, щоб здати лабораторну роботу, кожному студенту необхідно:

1. Отримати завдання у відповідності до варіанту;
2. Виконати його (самостійно);
3. Продемонструвати програму викладачеві;
5. Оформити звіт, перевірити його у викладача, захистити лабораторну роботу.
6. Зберегти файл, який містить звіт з лабораторної роботи, та файл-архів проекту в системі Moodle (сайт mentor.khai.edu).

Зміст звіту:

1. Номер варіанта;
2. Текст завдань;
3. Текст програми;
4. При необхідності – пояснення до реалізації;
5. Результати тестів (скріншоти).
6. Висновки.

Завдання до лабораторної роботи

Створення виконуваних програм.

До сих пір ми виконували програми на мові Haskell з використанням інтерпретатора. Однак існує можливість створювати окремі виконувані програми, для виконання яких не потрібне середовище інтерпретатора. Для цього використовується компілятор Glasgow Haskell Compiler, що викликається за допомогою команди ghc.

Для того, щоб скомпілювати набір модулів в виконувану програму, повинен бути визначений модуль з ім'ям Main, в якому необхідно визначити функцію main :: IO (). Цей модуль слід помістити в файл Main.hs. Для компіляції необхідно ввести в командному рядку наступну команду:

ghc make Main.hs

У разі, якщо програма містить помилки, інформація про них буде виведена на екран. Якщо помилок немає, компілятор створить виконуваний файл, який можна запускати на виконання.

Завдання. Частина 1.

1. Робота з типом Expr. Використовуючи тип Expr, визначений вище, реалізуйте наступні функції (використовуйте для тестування функцію parseExpr).

1) Визначте коректну функцію diff, яка приймає в якості додаткового аргументу ім'я змінної, по якій необхідно здійснювати диференціювання.

2) Визначте функцію simplify, яка спрощує вирази типу Expr, застосовуючи очевидні правила виду:

$x + 0 = 0 + x = x$
$x \cdot 1 = 1 \cdot x = x$
$x \cdot 0 = 0 \cdot x = 0$ *і т. д.*

3) Визначте функцію toString, перетворюючу вираз типу Expr в рядок. Наприклад, результатом застосування функції до вираження Add (Mult (Const 2) (Var "x")) (Var "y") повинна бути рядок "2 * x + y". Врахуйте можливість використання дужок, наприклад, вираз Mult (Const 2) (Add (Var "x") (Var "y")) має перетворюватися в рядок "2 * (x + y)".

4) Визначте функцію eval, яка приймає два параметри: вираз типу Expr і список пар типу (String, Integer), що задає відповідність імен змінних і їх значень. Функція повинна обчислювати значення вираз з урахуванням заданих значень виразів. Наприклад, вираз eval (Add (Var "x") (Var "y")) [("x", 1), ("y", 2)] має видавати число 3.

2. Функції для роботи з типом List. Для введеного раніше типу List визначте наступні функції:

1) lengthList, що повертає довжину списку типу List.

2) nthList, що повертає n-й елемент списку.

3) removeNegative, яка зі списку цілих (тип List Integer) видаляє негативні елементи.

4) fromList, перетворюючу список типу List в звичайний список.

5) toList, перетворюючу звичайний список в список типу List.

3. Функції роботи з бінарними деревами пошуку. Визначте тип даних, який представляє бінарні дерева пошуку. На відміну від дерев, представлених в методичних вказівках, в деревах пошуку дані можуть знаходитися не тільки в листі, але і в проміжних вузлах дерева. Будемо використовувати дерева для подання асоціативного масиву, що зіставляють значення ключів (подаються як рядки) цілих чисел. Для кожного вузла з деяким ключем в лівому піддереві повинні міститися елементи з меншими значеннями ключа, а в правому – з великими. При пошуку відповідності між рядком і числом необхідно враховувати цю інформацію, оскільки вона дозволяє більш ефективно отримувати інформацію з дерева. Визначте описаний тип даних і наступні функції:

1) add, що додає в дерево задану пару ключа та значення.

2) find, яка повертає число, відповідне заданому рядку.

3) exists, яка перевіряє, що елемент із заданим ключем міститься в дереві.

4) toList, що перетворює задане дерево пошуку в список, упорядкований за значеннями ключів.

4. Розробити тип даних, який представляє вміст каталогу файлової системи. Вважаємо, що кожен файл або містить певні дані, або є

каталогом. Каталог включає в себе інші файли (які, в свою чергу можуть бути каталогами) разом з їхніми іменами і розмірами в байтах. У даній роботі вміст файлів можна ігнорувати: тип даних повинен представляти тільки їх імена, розміри і структуру каталогів. Визначте наступні функції:

1) dirAll, що повертає список повних імен всіх файлів каталогу, включаючи підкаталоги.

2) find, яка повертає шлях, що веде до файлу з заданим іменем. Наприклад, якщо каталог містить файли a, b і c, і b є каталогом, що містить x та y, тоді функція пошуку для x повинна повернути рядок "b / x".

3) du, для заданого каталогу повертає кількість байт, займаних його файлами (включаючи файли в підкаталогах).

5. Розробіть тип даних Prop, що представляє затвердження такого виду:

Твердженням будемо називати логічну формулу, що має одну з наступних форм:

- ім'я змінної (рядок)
- p & q
- p | q
- ~ p

де p і q – твердження. Наприклад, твердженнями є такі формули:

- x
- x | y
- x & (x | ~ y)

Визначте наступні функції:

1) vars :: Prop -> [String], яка повертає список імен змінних (без повторень), що зустрічаються в твердженнях.

2) Нехай заданий список імен змінних і їх значень типу Bool, наприклад [("x", True), ("y", False)]. Визначте функцію truthValue :: Prop -> [(String, Bool)] -> Bool, яка визначає, чи вірно твердження, якщо змінні мають задані списком значення.

3) Визначте функцію tautology:: Prop -> Bool, котра повертає True, якщо твердження вірне при будь-яких значеннях змінних, що зустрічаються в ньому (наприклад, це виконується для твердження (x | ~ x)).

6. Лексичні дерева (trie-дерева) використовуються для подання словників. Кожен вузол дерева містить наступну інформацію: символ, булевське значення і список піддерев (у кожного вузла може бути будь-яка кількість дочірніх дерев).

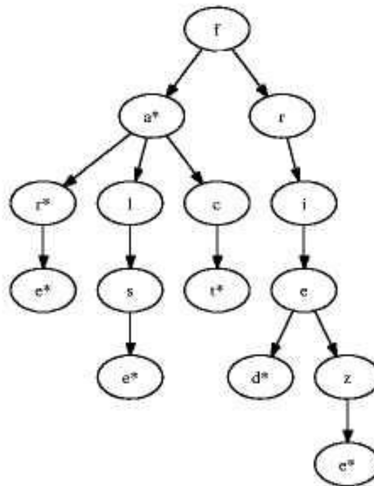


Рисунок 1 – trie-дерево

Приклад trie-дерева наведено на рисунку 1. Булевське значення, рівне True відзначає кінець слова, що читається, починаючи з кореня дерева. На малюнку вузли з такими значеннями позначені символом *. Таким чином, в дереві представлені слова fa, false, far, fare, fact, fried, frieze. Визначте наступні функції:

1) exists, яка перевіряє, що заданий слово міститься в trie-дереві.

2) insert, яка по дереву і слову повертає нове дерево, в яке включено це слово. Якщо слово вже міститься в дереві, воно повинно повертатися без змін.

3) completions, яка по заданому рядку повертає список всіх слів з дерева, початком яких служить зазначена строка (наприклад, для наведеного на рисунку 1 дерева по рядку "fri" повинен повертатися список ["fried", "frieze"]).

7. Теоретично можливо, хоча і неефективно, визначити цілі числа за допомогою рекурсивних типів даних в такий спосіб:

data Number = Zero | Next Number

Тобто число є або нулем (Zero), або визначається, як число, наступне за попереднім числом. Наприклад, число 3 записується як Next (Next (Next Zero)). Визначте для такого подання наступні функції:

1) fromInt, для заданого цілого числа типу Integer повертає відповідне йому значення типу Number.

2) toInt, перетворюючу значення типу Number у відповідне ціле число.

3) plus:: Number -> Number -> Number, складаються свої аргументи.

4) mult:: Number -> Number -> Number, множити свої аргументи.

5) dec, віднімається одиниця зі свого аргументу. Для Zero функція повинна повертати Zero.

6) fact, яка обчислює факторіал.

8. Ієрархія посад в деякій організації утворює деревоподібну структуру. Кожен працівник, однозначно характеризується унікальним ім'ям, має кілька підлеглих. Визначте тип даних, що представляє таку ієрархію і опишіть наступні функції:

1) getSubordinate, що повертає список підлеглих вказаного працівника.

2) getAllSubordinate, що повертає список всіх підлеглих даного працівника, включаючи непрямих.

3) getBoss, що повертає начальника зазначеного працівника.

4) getList, що повертає список пар, першим елементом яких є ім'я працівника, а другим – кількість його підлеглих (включаючи непрямих).

9. Область на площині є або прямокутником, або кругом, або об'єднанням областей, або їх перетином. Прямокутник характеризується координатами лівого нижнього і правого верхнього кутів, коло – координатами центру та радіусом. Розробіть структуру даних, що представляє область описаного виду. Визначте наступні функції:

1) contains, що перевіряє, що задана точка потрапляє в область.

2) isRectangular, що перевіряє, що область задається тільки прямокутниками.

3) isEmpty, що перевіряє, що область порожня,тобто жодна точка площини не потрапляє в неї.

10. Клас в об'єктно-орієнтованої мови містить набір методів (в даній роботі будемо ігнорувати поля-дані класу). Крім того, він може мати єдиний батьківський клас (НЕ роздивляємося випадок множинного успадкування). Однак існують класи і без батьків. При спадкуванні класу методи предка додаються до методів нащадка. Визначте тип даних, що представляє інформацію про ієрархію класів. Опишіть наступні функції:

1) getParent, що повертає безпосереднього предка класу з вказаним ім'ям.

2) getPath, що повертає список всіх предків даного класу (безпосередній предок, пращур предка і т. д.)

3) getMethods, що повертає список методів зазначеного класу з урахуванням наслідування.

4) inherit, що додає в ієрархію класів клас із заданим ім'ям, успадкований від зазначеного предка.

Завдання. Частина 2.

Кожен студент самостійно виконує завдання, згідно з номером варіанту в загальному списку (за журналом).

1. На площині задані n точок своїми координатами (x1, y1), (x2, y2), ... Скласти програму обчислення максимального внутрішнього і мінімального

зовнішнього радіусів кільця з центром на початку координат, що містить всі точки. Координати точок зчитувати з файлу.

2. Скласти програму, яка вводить натуральне число N і видає всі тризначні числа, сума цифр яких дорівнює.

3. Скласти програму, яка зчитує з командного рядка ім'я файлу і виводить на екран кількість символів, рядків і слів в цьому файлі.

4. Програма повинна зчитувати з командного рядка імена двох файлів і виводити на екран ті рядки цих файлів, які відрізняються один від одного.

5. Програма, яка приймає ім'я файлу і виводить його рядки, перед кожною з яких записується її номер.

6. Програма зчитує з файлу матрицю і виводить на екран суми стовпців цієї матриці.

7. Програма зчитує з командного рядка ім'я файлу і слово і перевіряє, чи зустрічається таке слово в файлі.

8. Програма приймає ім'я файлу, сортує його рядки і видає їх на екран.

9. Програма зчитує з командного рядка ім'я файлу і виводить на екран всі слова, що зустрічаються в цьому файлі, без повторень.

10. Програма зчитує з командного рядка ім'я файлу і виводить на екран найбільш часто зустрічається в ньому слово.

11. Скласти програму, яка запитує у користувача двозначне ціле число, вводить його і відображає на екрані величину числа словами. Наприклад, введено – 12. Результат: мінус дванадцять.

12. Програма приймає ім'я файлу і символ і виводить на екран ті рядки файлу, які починаються з вказаного символу.

13. Програма для перетворення одиниць вимірювання призначена для перетворення фізичних величин один в одного (метри в кілометри або сантиметри, кілограми в тонни, грами і фунти). Програма приймає в командному рядку число, вихідні одиниці і цільову одиницю виміру і виводить на екран перетворене число. Одиниці виміру задавати рядками («m» - для метрів, «km» - кілометри, «cm» - сантиметри, «kg» - кілограми, «t» - тонни, «g» - грами, «p» - фунти).

14. Програма, яка за заданою кількістю N друкує список всіх простих чисел, що не перевищують N. Програма, яка за заданою кількістю N друкує список всіх скоєних чисел, що не перевищують N. Досконалим називається число, яке дорівнює сумі свої подільників, наприклад $6 = 1 + 2 + 3$ *або* $24 = 1 + 2 + 3 + 4 + 6 + 8$.

15. Програма зчитує з командного рядка ім'я файлу і два слова і роздруковує на екрані вміст файлу, в якому все входження першого слова замінені на друге.

16. Програма, яка виводить на екран всі алфавітно-цифрові послідовності, що містяться в зазначеному файлі. Алфавітно-цифровий називається безперервна послідовність цифр і букв латинського алфавіту.

Приклад: якщо файл містить символи asdf - + ^ & @ qwert1y! @ # то результатом роботи програми має бути:

asdf

qwert1y

17. Програма приймає в командному рядку ім'я файлу і один з двох символів 'u' або 'l'. Залежно від того, який із символів переданий, вона перетворює вміст файлу до верхнього або нижнього регістру і роздруковує на екрані.

18. Скласти програму, яка вводить натуральне число N і підстава системи числення m, а потім виводить цифри уявлення N в m-річної системі числення.

19. Програма зчитує з файлу матрицю і виводить на екран суму діагональних елементів цієї матриці.

20. Програма зчитує дві матриці з файлів і записує в третій файл матрицю, що є їхньою сумою.

21. Програма зчитує з одного файлу матрицю, а з іншого – вектор і записує в третій файл результат множення матриці на вектор.

22. Файл містить записи наступного формату: операція число, де <операція> - одне з двох слів «inc» або «dec», а <число> представляє собою деяке ціле число. Програма повинна по заданому файлу з такими записами перевірити, що сума чисел, відповідних слову «inc» дорівнює сумі чисел для «dec».

Розрахункова робота. «Функціональні мови програмування: застосування та приклади».

Проведення демо за підсумками відповідної ітерації.

Мета розрахункової роботи – провести аналіз актуальності використання функціональних мов програмування.

Порядок виконання роботи:

1. Тему розрахункової роботи можна вибрати самостійно. Основна вимога – продемонструвати сучасні підходи до розробки програмного забезпечення (технології, методи, інструментальні засоби, фреймворки) на реальних існуючих проектах з використанням мови програмування F# (або будь-якої іншої мови функціональної парадигми програмування).
2. Кожна тема домашнього завдання – унікальна. Варіанти вибрати і розділити між собою самостійно.
3. Тобто або вибираємо актуальний популярний проект і описуємо використовувані при його розробці парадигми програмування, або, навпаки, для початку визначаємося з мовою розробки програмного забезпечення, а потім показуємо, в яких проектах застосовується і для досягнення яких цілей.
4. Оформити звіт, перевірити у викладача і захистити роботу.
5. Зберегти файл, який містить звіт з розрахункової роботи в системі Mentor (mentor.khai.edu).

Зміст звіту:

РР повинна містити:

- титульний лист;
- зміст;
- постановку завдання;
- теоретичні відомості з обраної тематики;
- приклади використання обраної технології (методу, фреймворка, середовища розробки): в яких існуючих проектах використовувалися до поточного моменту часу, перспективи використання, приклади практичної реалізації (наприклад, доступні на веб-хостингах або opensourse);
- висновки з розрахункової роботи.

РР має бути оформлено відповідно до вимог «Нормоконтролер» (методичні рекомендації В. М. Павленко).

ПИТАННЯ, ТЕСТИ ДЛЯ КОНТРОЛЬНИХ ЗАХОДІВ

Тема 1. Вступ до функціонального програмування.

1. Історія розвитку функціонального програмування (ФП).
2. Сучасний стан теорії ФП.
3. Переваги та недоліки функціонального підходу.
4. Основні властивості функціональних мов програмування.
5. Приклади функціональних мов програмування.
6. Типові задачі, що вирішуються методами ФП.

Тема 2. Базові принципи функціональної мови програмування Haskell.

1. Поняття списку у функціональному програмуванні.
2. Базисні операції для роботи зі списками.
3. Списки й облікові структури.
4. Програмна реалізація списків у функціональних мовах.
5. Списки в мові Haskell. Генератори списків і математичні послідовності. Кортежі.
6. Функція – основний об’єкт функціонального програмування.
7. Умови по іменуванню об’єктів у мові Haskell.
8. Опис і визначення функцій мовою Haskell.
9. Передача параметрів і повернення значень функціями.
10. Виклики функцій. Використання лямбда-обчислень. Функція як об’єкт для передачі в інші функції.
11. Структури й типи даних. Синоніми типів. Типи функцій у функціональних мовах. Поліморфні типи. Часткове застосування. Ледачі (відкладені) обчислення мовою Haskell.
12. Охоронні вирази і конструкції. Розгалуження алгоритму.
13. Двовимірний синтаксис. Локальні змінні для оптимізації коду функціональною мовою й мовою Haskell.
14. Використання накопичуючого параметра (акумулятора) для оптимізації процесу обчислень.
15. Головна й хвостова рекурсії.
16. Модулі й абстрактні типи даних. Модулі як способи структуризації й організації програм мовою Haskell.
17. Імпорт і експорт даних за допомогою модулів.
18. Приховування даних. Абстрактні типи даних і інтерфейси.

Тема 3. Класи та їхні екземпляри у мові Haskell.

1. Симбіоз парадигм функціонального й об’єктно-орієнтованого програмування. Підтримка мовою Haskell об’єктно-орієнтованих механізмів й методів.

2. Параметричний поліморфізм даних. Поняття класу і його реалізації в мові Haskell. Приклади параметричного поліморфізму в імперативних і функціональних мовах, а також у мові Haskell.

3. Класи в мові Haskell як спосіб абстракції дійсності. Розширений опис поняття класу в мові Haskell.

4. Методи класу – шаблони функцій для реалізації обробки даних. Мінімальний опис методів класу й зв'язок методів. Визначення класів.

5. Спадкування й реалізація. Спадкування класів і спадкування методів. Екземпляри класів – реалізація інтерфейсів, надаваних реалізованим класом. Реалізація методів для обробки даних. Клас – шаблон типу, реалізація класу – тип даних. Реалізація для існуючих типів. Сорти типів.

6. Стандартні класи мови Haskell. Короткий опис всіх стандартних класів, розроблених для полегшення програмування мовою Haskell.

7. Дерево спадкування стандартних класів. Типові способи використання стандартних класів мови Haskell. Реалізація стандартних класів – типи в мові Haskell.

8. Порівняння з іншими мовами програмування. Порівняння понять «клас» і «реалізація класу» у мові Haskell з об'єктно-орієнтованими мовами програмування (на прикладі мов C+ + і Java, а також деяких інших мов). Глобальні відмінності поняття «клас» у функціональних і об'єктно-орієнтованих мовах.

Тема 4. Комбінаторна логіка й λ-обчислення.

1. Комбінатори й обчислення за допомогою комбінаторів. Базиси в комбінаторній логіці.

2. Використання базисних комбінаторів для вираження будь-яких обчислювальних процесів.

3. Числа й інші математичні об'єкти у вигляді комбінаторів. Базові комбінатори.

4. Абстракція функцій як обчислювальних процесів. Функція – об'єкт математичного дослідження. Обчислювальний процес – функція. Опис функцій як лямбда-виражень.

5. Вільні, і зв'язані ідентифікатори. Застосування (аплікація) значень до лямбда-виражень. Редукція. Теза Черча-Тьюринга.

6. Лямбда-обчислення як теоретична основа функціонального програмування.

7. Інтенсіонал і екстенсіонал функцій. Правила виводу.

8. Відповідності між обчисленнями функціональних програм і редукцією лямбда-виражень.

9. Кодування даних в лямбда-обчислення. Механізм кодування даних в лямбда-обчисленні. Редукція й обчислення у функціональних мовах.

10. Поняття редукції. Часткові обчислення з погляду редукції лямбда-виражень.

11. Стратегія редукції й стратегія обчислень. Ледача редукція.

ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

(мова F#)

1. Визначення та коротка історія функціонального програмування.
2. Абстракція і декомпозиція. Декларативне програмування.
3. Парадигми програмування.
4. Функціональне програмування в реальному житті. Реальні приклади застосування.
5. Основні принципи функціонального програмування.
6. Зіставлення зі зразком. Рекурсія. Цикли.
7. Приклад побудови графіка 2D-функції.
8. Рекурсивні структури даних. Списки.
9. Приклади роботи зі списками.
10. Хвостова рекурсія. Порядкове подання списків і матриць.
11. Функціональні структури даних.
12. Дерева. Дерева виразів і дерева пошуку.
13. Введення в л-числення.
14. Нормальний і аплікативний порядки редукції. Теорема Черча-Россера.
15. Опис рекурсивних функцій. Комбінатори і комбінаторна логіка.
16. Від л-обчислення до мови програмування.
17. Замикання, генератори та відкладені обчислення.
18. Послідовності та ледачі обчислення в F #. Мемоізація.
19. Приклад реалізації машини Тьюринга.
20. Типізація в мовах функціонального програмування.
21. Формальна семантика мов функціонального програмування.
22. Доказ властивостей програм.
23. Реалізація функціональних мов. Eval-Apply-інтерпретатори.
24. Реалізація функціональних мов – інтерпретатори й абстрактні машини.
25. Реалізація функціональних мов – редукція графів, потокові реалізації.
26. Аналіз штучних і природних мов.
27. Метапрограмування – Ктатіонс.
28. Імперативне ядро у функціональних мовах. Монада. Computational Workflows.
29. Асинхронні і паралельні обчислення.

(мова Haskell)

30. У чому відмінність команд інтерпретатора від виразів мови Haskell?
31. Основні типи мови Haskell.
32. Функції для роботи з кортежами.
33. Функції для роботи зі списками.
34. Допустимі імена змінних і функцій.
35. Команди інтерпретатора для роботи з файлами програм.
36. Умовні вирази в мові Haskell.
37. Визначення функцій в мові Haskell.
38. Правила вирівнювання.
39. Зіставлення зі зразком.
40. Операція вибору.
41. Кусочне завдання функцій.
42. Визначення локальних змінних.
43. Умови, які охороняють.
44. Поліморфізм.
45. Визначення власних типів даних.
46. Лямбда-абстракції.
47. Операційні секції.
48. Функції вищих порядків.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Душкин Р. В. Функциональное программирование на языке Haskell // Р. В. Душкин. – М. : ДМК Пресс, 2009. – 608 с.
2. Роганова Н. А. Функциональное программирование: учеб. пособ. для студентов высших учебных заведений // Н. А. Роганова. – М. : ГИНФО, 2012. – 260 с.
3. Филд А. Функциональное программирование // А. Филд, П. Харрисон. – М. : Мир, 2013. – 638 с.
4. Городняя Л. В. Основы функционального программирования. Курс лекций – М. : Интернет-университет информационных технологий, 2014. – 280 с.
5. Филд А., Харрисон П. Функциональное программирование = Functional Programming. – М. : Мир, 2013. – 637 с.
6. John Harrison. Функциональное программирование. Курс лекций = Functional Programming. – 2007.
7. http://www.intuit.ru/studies/courses/29/29/info
8. Говорят, Haskell — язык для гениев и академиков. Правда? Haskell,
9. Функциональное программирование, Интервью https://habr.com/ru/post/438970/
10. Haskell — язык, позволяющий глубже понять программирование. Как он устроен и почему его выбирают разработчики? https://ru.hexlet.io/blog/posts/haskell-yazyk-pozvolyayuschiy-glubzhe-ponyat-programmirovanie-kak-on-ustroen-i-pochemu-ego-vybirayut-razrabotchiki
11. https://www.haskell.org/
12. Основные принципы программирования: функциональное программирование https://tproger.ru/translations/functional-programming-concepts/
13. Жаргон функционального программирования https://habr.com/ru/post/310172/
14. Функциональное программирование с примерами на JavaScript. Часть первая. Основные техники функционального программирования https://tproger.ru/translations/functional-js-1/
15. https://docs.microsoft.com/ru-ru/dotnet/fsharp/

СЛОВНИК ТЕРМІНІВ З ПРИКЛАДАМИ

«Жаргон» функціонального програмування

Arity (арність).

Кількість аргументів функції. Від слів унарний, бінарний, тернарний (unary, binary, ternary) і так далі. Це незвичайне слово, тому що складається з двох суфіксів: "-ary" і "-ity.". Додавання, наприклад, приймає два аргументи, тому це бінарна функція, або функція, у якій арность дорівнює двом. Іноді використовують термін "Діадне" (dyadic), якщо вважають за краще грецьке коріння замість латинських. Функція, яка приймає будь-яку кількість аргументів називається, відповідно, варіативної (variadic). Але бінарна функція може приймати два і тільки два аргументи, без урахування каррінг або часткового застосування.

```
const sum = (a, b) => a + b
const arity = sum.length
console.log (arity)// 2
// The arity of sum is 2
```

Higher-Order Functions (функції вищого порядку).

Функція, яка приймає функцію як аргумент і / або повертає функцію.

```
const filter = (predicate, xs) => {
  const result = []
  for (let idx = 0; idx <xs.length; idx ++) {
  if (Predicate (xs [idx])) {
  result.push (xs [idx])
  }
  }
  return result
}
```

```
const is = (type) => (x) => Object(X) instanceof type
```

```
filter (is(Number), [0, '1', 2, null]) // [0, 2]
```

Partial Application (часткове застосування).

Часткове застосування функції означає створення нової функції з пред-заповненням деяких аргументів оригінальної функції.

```
// Helper to create partially applied functions
// Takes a function and some arguments
const partial = (f, ... args) =>
  // returns a function that takes the rest of the arguments
  (... moreArgs) =>
  // and calls the original function with all of them
  f (... args, ... moreArgs)
// Something to apply
const add3 = (a, b, c) => a + b + c
// Partially applying `2` and` 3` to `add3` gives you a one-argument function
const fivePlus = partial (add3, 2, 3) // (c) => 2 + 3 + c
fivePlus (4) // 9
```

Також в JS можна використовуватиFunction.prototype.bindдля часткового застосування функції:

```
const add1More = add3.bind (null, 2, 3) // (c) => 2 + 3 + c
```

Завдяки попередній підготовці даних часткове застосування допомагає створювати більш прості функції з більш складних. Функції з каррінг автоматично виконують часткове застосування.

Currying (каррінг).

Процес конвертації функції, яка приймає кілька аргументів, в функцію, яка приймає один аргумент за раз.

При кожному виклику функції вона приймає один аргумент і повертає функцію, яка приймає один аргумент до тих пір, поки всі аргументи не будуть оброблені.

```
const sum = (a, b) => a + b
```

```
const curriedSum = (a) => (b) => a + b
```

```
curriedSum (40) (2) // 42.
```

```
const add2 = curriedSum (2) // (b) => 2 + b
```

```
add2 (10) // 12
```

Auto Currying (автоматичний каррінг).

Трансформація функції, яка приймає кілька аргументів, в нову функцію. Якщо в нову функцію передати менше ніж передбачено кількість аргументів, то вона поверне функцію, яка приймає залишилися аргументи. Коли функція отримує правильну кількість аргументів, то вона виконується.

У Underscore, lodash і ramda є функція curry.

```
const add = (x, y) => x + y
```

```
const curriedAdd = _.curry (add)
```

```
curriedAdd(1, 2) // 3
```

```
curriedAdd(1) // (y) => 1 + y
```

```
curriedAdd(1) (2) // 3
```

Function Composition (композиція функцій).

З'єднання двох функцій для формування нової функції, в якій висновок першої функції є введенням другої.

```
const compose = (f, g) => (a) => f (g (a)) // Definition
```

```
const floorAndToString = compose ((val) => val.toString (), Math.floor)// Usage
```

```
floorAndToString (121.212121) // '121'
```

Purity (чистота).

Функція є чистою, якщо повертається їй значення визначається виключно вступними значеннями, і функція не має побічних ефектів.

```
const greet = (name) => 'Hi,' + name
```

```
greet ('Brianne') // 'Hi, Brianne'
```

На відміну від:

```
let greeting
```

```
const greet = () => {
```

```
 greeting = 'Hi,' + window.name
```

```
}
```

```
greet ()// "Hi, Brianne"
```

Side effects (побічні ефекти).

У функції є побічні ефекти, якщо крім повернення значення вона взаємодіє (читає або пише) з зовнішнім змінним станом.

```
const differentEveryTime = new Date()
```

```
console.log ('IO is a side effect! ')
```

Idempotent (ідемпотентність).

Функція є ідемпотентною, якщо повторне її виконання викликає такий же результат.

```
f(F (x))≍ f(X)
```

```
Math.abs(Math.abs(10))
```

```
sort (sort(sort([2, 1])))
```

Point-Free Style (безточечна нотація).

Написання функцій в такому вигляді, що визначення не явно вказує на кількість використовуваних аргументів. Такий стиль зазвичай вимагає каррінг або іншої функції вищого порядку (або в цілому – неявного програмування).

```
// Given
const map = (fn) => (list) => list.map (fn)
const add = (a) => (b) => a + b
// Then
// Not points-free - `numbers` is an explicit argument
const incrementAll = (numbers) => map (add (1)) (Numbers)
// Points-free - The list is an implicit argument
const incrementAll2 = map (add (1))
```

Функція incrementAll визначає і використовує параметр numbers, так що вона не використовує бесточечную нотацію. incrementAll2 просто комбінує функції і значення, не згадуючи аргументів. Вона використовує бесточечную нотацію.

Визначення з бесточечной нотацією виглядають як звичайні привласнення без function або =>.

Predicate (предикат).

Предикат – це функція, яка повертає true або false залежно від переданого значення. Поширений випадок використання предиката – функція зворотного виклику (callback) для фільтра масиву.

```
const predicate = (a) => a> 2
; [1, 2, 3, 4] .Filter (predicate) // [3, 4]
```

Categories (категорії).

Об'єкти з функціями, які підпорядковуються певним правилам. Наприклад, моноїд.

Value (значення).

Все, що може бути присвоєно змінній.

```
5
```

```
Object.freeze({name: 'John', Age: 30}) // The `freeze` function enforces immutab
```

```
ility.
```

```
; (A) => a
```

```
; [1]
```

```
undefined
```

Constant (константа).

Змінна, яку не можна перепризначити після визначення.

```
const five = 5
```

```
const john = {name: 'John', Age: 30}
```

Константи мають референциальной прозорістю або прозорістю посилань (referential transparency). Тобто, їх можна замінити значеннями, які вони представляють, і це не вплине на результат.

З константами з попереднього лістингу такий вираз вище завжди буде повертатитrue.

```
john.age + five === ({name: 'John', Age: 30}). Age + (5)
```

Functor (функтор).

Об'єкт, який реалізує функціюmap, Яка при проході за всіма значеннями в об'єкті створює новий об'єкт, і підпорядковується двом правилам:

```
// зберігає нейтральний елемент (identity)
```

```
object.map (x => x) ===object
```

і

```
// підтримує композицію
```

```
object.map(X => f (g (x))) === object.map(G).map(F)
```

(f, g – довільні функції).

В JavaScript є функтор Array, Тому що він підпорядковується ці правилам:

```
[1, 2, 3] .Map (x => x) // = [1, 2, 3]
```

і

```
const f = x => x + 1
```

```
const g = x => x * 2
```

```
; [1, 2, 3] .Map (x => f (g (x))) // = [3, 5, 7]
```

```
; [1, 2, 3] .Map (g) .map (f) // = [3, 5, 7]
```

Pointed Functor (вказує на функтор).

Об'єкт з функцією of з будь-яким значенням. У ES2015 є Array.of, що робить масиви вказівним функтором.

```
Array.of(1) // [1]
```

Lift.

Lifting – це коли значення поміщається в об'єкт на зразок функтора. Якщо "підняти" (lift) функцію в аплікативного функтор, то можна змусити її працювати зі значеннями, які також присутні в функтором.

У деяких реалізаціях є функція lift або liftA2, які використовуються для спрощення запуску функцій над функтором.

```
const liftA2 = (f) => (a, b) => a.map (f) .ap (b)
```

```
const mult = a => b => a * b
```

```
const liftedMult = liftA2 (mult) // this function now works on functors like arr
ay
liftedMult ([1, 2], [3]) // [3, 6]
liftA2 ((a, b) => a + b) ([1, 2], [3, 4]) // [4, 5, 5, 6]
```

Підйом функції з одним аргументом і її застосування виконує те ж саме, що і map.

```
constincrement = (X) => x + 1
lift (increment) ([2]) // [3]
; [2] .Map (increment) // [3]
```

Referential Transparency (прозорість посилань).

Якщо вираз можна замінити його значенням без впливу на поведінку програми, то воно володіє прозорістю посилань.

Наприклад, є функція greet:

```
const greet = () =>'Hello World!'
```

будь який виклик greet() можна замінити на Hello World!, так що ця функція є прозорою (referentially transparent).

Lambda (лямбда).

Анонімна функція, яку можна використовувати як значення.

```
; (function (A) {
  return a + 1
})
; (A) => a +1
```

Лямбда часто передають в якості аргументів у функції вищого порядку.

```
[1, 2] .Map ((a) => a +1) // [2, 3]
```

Лямбда можна привласнити змінної.

```
const add1 = (a) => a +1
```

Lambda Calculus (лямбда-числення).

Область інформатики, в якій функції використовуються для створення універсальної моделі обчислення.

Lazy evaluation (ледачі обчислення).

Механізм обчислення при необхідності, із затримкою обчислення виразу до того моменту, поки значення не буде потрібно. У функціональних мовах це дозволяє створювати структури на кшталт нескінченних списків, які в звичайних умовах неможливі в імперативних мовах програмування, де черговість команд має значення.

```
const rand = function* () {
 while (1 < 2) {
 yield Math.random ()
 }
}
const randIter = rand ()
randIter.next ()// Кожен виклик дає випадкове значення, вираз виповнюється при н
еобхідності.
```

Monoid (моноїд).

Об'єкт з функцією, яка "комбінує" об'єкт з іншим об'єктом того ж типу. Простий приклад моноїд це складання чисел:

```
1 + 1 // 2
```

У цьому випадку число – це об'єкт, а + це функція.

Повинен існувати нейтральний елемент (identity), так, щоб комбінування значення з ним не змінювало значення. У разі складання таким елементом є 0.

```
1 + 0 // 1
```

Також необхідно, щоб угруповання операцій не впливала на результат (асоціативність):

```
1 + (2 + 3) === (1 + 2) + 3 // true
```

Конкатенація масивів – це теж моноїд:

```
; [1, 2] .Concat ([3, 4]) // [1, 2, 3, 4]
```

Нейтральний елемент – це порожній масив[]

```
; [1, 2] .Concat ([])// [1, 2]
```

Якщо існують функції нейтрального елемента і композиції, то функції в цілому формують моноїд:

```
const identity = (a) => A
```

```
const compose = (f, G) => (x) => F (g(x))
```

foo – це будь-яка функція з одним аргументом.

```
compose (foo,identity) ≍ compose (identity, Foo) ≍ foo
```

Monad (монада).

Монада – це об'єкт з функціямиofichain.chaincхожий намap, Але він виробляє розкладання вкладених об'єктів в результаті.

```
// Implementation
```

```
Array.prototype.chain = function (f) {
```

```
return this.reduce ((acc, it) => acc.concat (f (it)), [])}
```

```
// Usage
```

```
; Array.of('Cat, dog', 'Fish, bird') .Chain ((a) => a.split (',')) // ['Cat', 'D
og', 'Fish', 'Bird']
// Contrast to map
; Array.of('Cat, dog', 'Fish, bird') .Map ((a) => a.split (',')) // [['Cat', 'Do
g'], ['Fish', 'Bird']]
```

of також відомий як returnв інших функціональних мовах. chain також відомий як flat map і bind в інших мовах.

Comonad (комонада).

Об'єкт з функціями extract і extend.

```
const CoIdentity = (v) => ({
  val: V,
  extract () {
  return this.val
  },
  extend (f) {
  return CoIdentity (f (this))
  }
})
```

Extract бере значення з функтора.

```
CoIdentity(1) .Extract () // 1
```

Extend виконує функцію на комонаде. Функція повинна повернути той же тип, що комонада.

```
CoIdentity(1) .Extend ((co) => co.extract () + 1) // CoIdentity (2)
```

Applicative Functor (аплікативний функтор).

Об'єкт з функцієюap.арзастосовує функцію в об'єкті до значення в іншому об'єкті того ж типу.

```
// Implementation
Array.prototype.ap =function (Xs) {
 return this.reduce ((acc, f) => acc.concat (xs.map (f)), [])
}
// Example usage
; [(A) => a +1] .Ap ([1]) // [2]
```

Це корисно, коли є два об'єкти, і потрібно застосувати бінарну операцію на їх вмісті.

```
// Arrays that you want to combine
const arg1 = [1, 3]
const arg2 = [4, 5]
// combining function - must be curried for this to work
const add = (x) => (y) => x + y
const partiallyAppliedAdds = [add] .ap (arg1) // [(y) => 1 + y, (y) => 3 + y]
```

В результаті отримаємо масив функцій, які можна викликати зарщоб отримати результат:

```
partiallyAppliedAdds.ap (arg2) // [5, 6, 7, 8]
```

Morphism (морфізм).
Функція трансформації.

Endomorphism (ендоморфізм).
Функція, у якій введення і виведення – одного типу.

```
// uppercase :: String -> String
const uppercase = (str) => str.toUpperCase ()
// decrement :: Number -> Number
const decrement = (x) => x - 1
```

Isomorphism (ізоморфізм).

Пара структурних трансформацій між двома типами об'єктів без втрати даних.

Наприклад, двовимірні координати можна зберігати в масиві [2, 3] або об'єкті{X: 2, y: 3}.

```
// Providing functions to convert in both directions makes them isomorphic.
const pairToCoords = (pair) => ({x: Pair [0], y: Pair [1]})
const coordsToPair = (coords) => [coords.x, coords.y]
coordsToPair (pairToCoords ([1, 2])) // [1, 2]
pairToCoords (coordsToPair ({x: 1, y: 2})) // {x: 1, y: 2}
```

Setoid.

Об'єкт, у якого є функція equals, яку можна використовувати для порівняння об'єктів одного типу.

Зробити масив сетоідом:

```
Array.prototype.equals = (Arr) => {
  const len = this.length
  if (Len! == arr.length) {
```

```
  return false
  }
  for (let i = 0; i <len; i ++) {
  if (this[I]! == arr [i]) {
  return false
  }
  }
  return true
}
; [1, 2].equals([1, 2]) // true
; [1, 2].equals([0]) // false
```

Semigroup (півгрупа).

Об'єкт з функцією concat, яка комбінує його з іншим об'єктом того ж типу.

```
; [1] .Concat ([2]) // [1, 2]
```

Foldable.

Об'єкт, в якому є функція reduce, яка трансформує об'єкт в інший тип.

```
const sum = (list) => list.reduce ((acc, val) => acc + val,0)
sum ([1, 2, 3]) // 6
```

Type Signatures (сигнатури типу).

Часто функції в JavaScript містять коментарі з зазначенням типів їх аргументів і значень. У співтоваристві існують різні підходи, але вони все схожі:

```
// functionName :: firstArgType -> secondArgType -> returnType
```

```
// add :: Number -> Number -> Number
```

```
const add = (x) => (y) => x + y
```

```
// increment :: Number -> Number
```

```
const increment = (x) => x + 1
```

Якщо функція приймає іншу функцію як аргумент, то її поміщають в дужки.

```
// call :: (a -> b) -> a -> b
```

```
const call = (f) => (x) => f (x)
```

символи a, b, c, d показують, що аргументи можуть бути будь-якого типу.

Наступна версія функції map приймає:

1. функцію, яка трансформує значення типу a в інший тип b
2. масив значень типу a,

і повертає масив значень типу b.

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

```
const map = (F) => (list) => list.map(F)
```

Union type (тип-об'єднання).

Комбінація двох типів в один, новий тип.

В JavaScript немає статичних типів, але давайте уявимо, що ми винайшли тип NumOrString, Який є складаннямStringiNumber.

операція + в JavaScript працює з рядками і числами, так що можна використовувати наш новий тип для опису його введення і виведення:

```
// add :: (NumOrString, NumOrString) -> NumOrString
```

```
const add = (a, b) => a + b
```

```
add (1, 2) // повертає число 3
add ('Foo', 2) // повертає рядок "Foo2"
add ('Foo', 'Bar') // повертає рядок "FooBar"
```

Тип-об'єднання також відомо як алгебраїчний тип, розмічене об'єднання і тип-сума.

Product type (тип-виріб).

Тип-виріб комбінує типи таким способом:

```
// point :: (Number, Number) -> {x: Number, y: Number}
const point = (x, y) => ({x: x, y: y})
```

Його називають твором, тому що можливе значення структури даних цей твір (product) різних значень.

Option (опціон).

Тип-об'єднання з двома випадками: Some і None. Корисно для композиції функцій, які можуть не повертати значення.

```
// Naive definition
const Some = (v) => ({
  val: v,
  map (f) {
  return Some (f (this.val))
  },
  chain (f) {
  return f (this.val)
  }
})
const None = () => ({
  map (f) {
```

```
    return this
    },
    chain (f) {
    return this
    }
  })
  // maybeProp :: (String, {a}) -> Option a
  const maybeProp = (key, obj) => typeof obj [key] === 'Undefined'? None (): Some
  (obj [key])
```

використовуйте chain для побудови послідовності функцій, які повертають Option.

```
// getItem :: Cart -> Option CartItem
const getItem = (cart) => maybeProp ('Item', Cart)
// getPrice :: Item -> Option Number
const getPrice = (item) => maybeProp ('Price', Item)
// getNestedPrice :: cart -> Option a
const getNestedPrice = (cart) => getItem (obj) .chain (getPrice)
getNestedPrice ({})// None ()
getNestedPrice ({item: {foo: 1}}) // None ()
getNestedPrice ({item: {price: 9.99}}) // Some (9.99)
```

ПРИКЛАДИ ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ МОВОЮ F#

Лабораторна робота № 1.

Варіант 20

1. Результат роботи програми

На рисунку 1 зображено результат роботи програми.

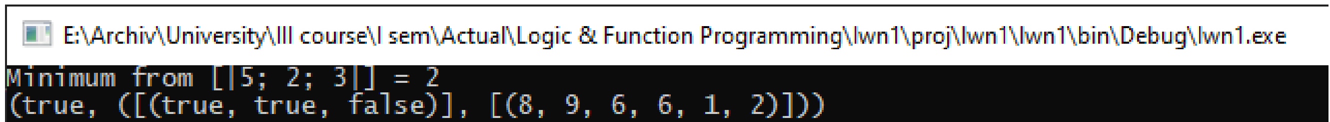


Рисунок 1 – Результат роботи програми

2. Лістинг програмного коду

Program.fs

```
let arr = [|5;2;3|]

let min3 items =
    items |> Array.min

//10)  (Bool,([Bool],[Integer]))
let task2 = true, ([true, true, false], [8,9,6,6,1,2])

[<EntryPoint>]
let main args =
    printf "%s" ("Minimum from ")
    printf "%A" (arr)
    printf "%s" (" = ")
    printfn "%A" (min3 (arr))
    printfn "%A" (task2)
    Console.ReadKey() |> ignore
    0
```

Лабораторна робота № 2.

Варіант 20

1. Результат роботи програми

На рисунку 2 зображено результат роботи програми.

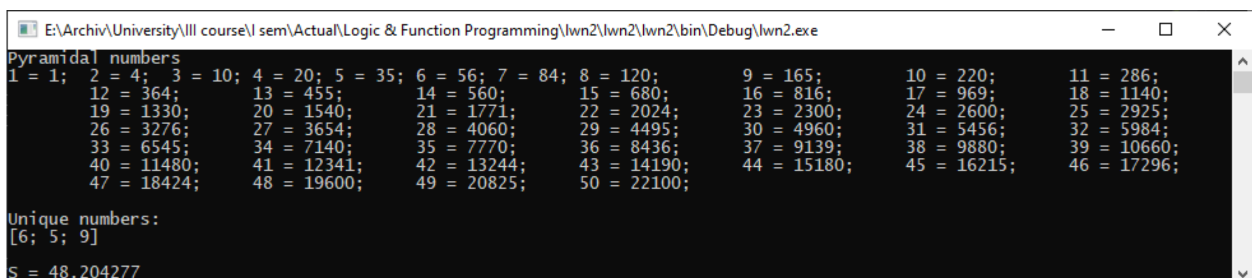


Рисунок 2 – Результат роботи програми

2. Лістинг програмного коду

Program.fs

```
// task1
let N = 50

// функция вычисления n-го треугольного числа
let t n = n * (n + 1) / 2

// рекурсивная функция вычисления n-го пирамидального числа
let rec p n =
    if n < 2 then 1
    else t(n) + p(n - 1)

// вывод первых n пирамидальных чисел
let printPyrNums n =
    printfn "Pyramidal numbers"
    for i in 1 .. N do
        printf "%d = %d;\t" i (p i)

//task 2
let myList = [7;6;8;7;8;5;7;3;9;3]

let count value list =
    List.where (fun y -> y = value) list
    |> List.length

let getUniqueList list = list |> List.filter (fun x -> count x list < 2)

//task3
let K = 3
let x = 5

let getLn3Pow(k) = log 3. ** k
let getPow(num, k) = num ** k
let rec getFact(k) =
    if k = 0 then
        1
    else
        k * getFact(k - 1)

let summa a b = a + b

let getS(n, k) =
    let mutable res = 0.
    for i in 1..k do
        let ln3 = getLn3Pow (float(i))
        let f = float(getFact i)
        let p = getPow(float(n), float(i))
        res <- summa res ((ln3 / f) * (p))
    res

[<EntryPoint>]
let main argv =
    printPyrNums N

    let uniqueNums = getUniqueList myList
    printfn "\n\nUnique numbers:\n%A" uniqueNums
```

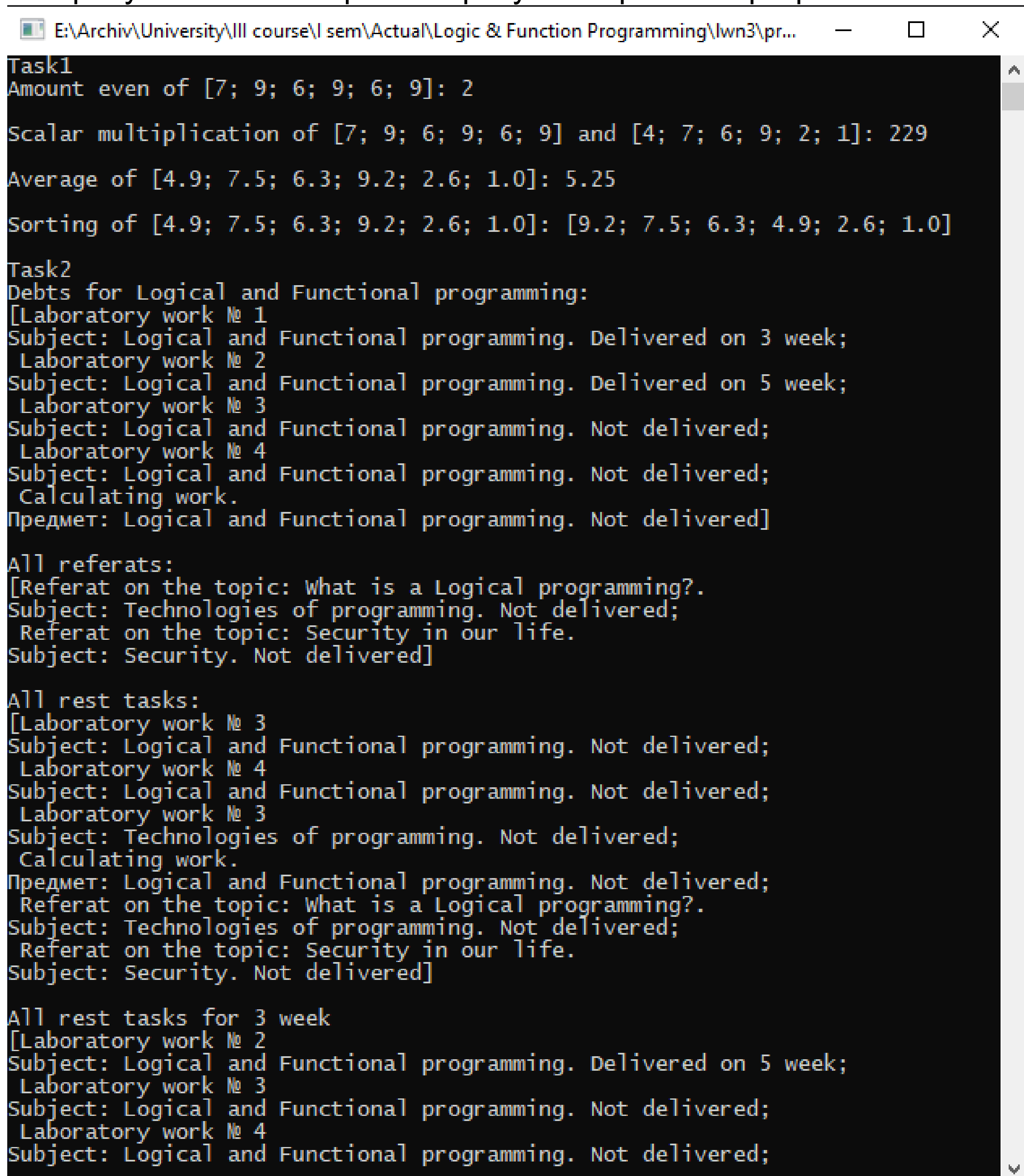
```
let S = getS(x,K)
printf "\nS = %f" S

System.Console.ReadKey() |> ignore
0 // return an integer exit code
```

Лабораторна робота № 3.

Варіант 20

1. Результат роботи програми

На рисунках 3 – 4 зображено результат роботи програми.



```
E:\Archiv\University\III course\I sem\Actual\Logic & Function Programming\lwn3\pr...
Task1
Amount even of [7; 9; 6; 9; 6; 9]: 2

Scalar multiplication of [7; 9; 6; 9; 6; 9] and [4; 7; 6; 9; 2; 1]: 229

Average of [4.9; 7.5; 6.3; 9.2; 2.6; 1.0]: 5.25

Sorting of [4.9; 7.5; 6.3; 9.2; 2.6; 1.0]: [9.2; 7.5; 6.3; 4.9; 2.6; 1.0]

Task2
Debts for Logical and Functional programming:
[Laboratory work № 1
Subject: Logical and Functional programming. Delivered on 3 week;
 Laboratory work № 2
Subject: Logical and Functional programming. Delivered on 5 week;
 Laboratory work № 3
Subject: Logical and Functional programming. Not delivered;
 Laboratory work № 4
Subject: Logical and Functional programming. Not delivered;
 Calculating work.
Предмет: Logical and Functional programming. Not delivered]

All referats:
[Referat on the topic: What is a Logical programming?.
Subject: Technologies of programming. Not delivered;
 Referat on the topic: Security in our life.
Subject: Security. Not delivered]

All rest tasks:
[Laboratory work № 3
Subject: Logical and Functional programming. Not delivered;
 Laboratory work № 4
Subject: Logical and Functional programming. Not delivered;
 Laboratory work № 3
Subject: Technologies of programming. Not delivered;
 Calculating work.
Предмет: Logical and Functional programming. Not delivered;
 Referat on the topic: What is a Logical programming?.
Subject: Technologies of programming. Not delivered;
 Referat on the topic: Security in our life.
Subject: Security. Not delivered]

All rest tasks for 3 week
[Laboratory work № 2
Subject: Logical and Functional programming. Delivered on 5 week;
 Laboratory work № 3
Subject: Logical and Functional programming. Not delivered;
 Laboratory work № 4
Subject: Logical and Functional programming. Not delivered;
```

Рисунок 3 – Результат роботи програми

```
Выбрать E:\Archiv\University\III course\I sem\Actual\Logic & Function Programming\lwn3\proj\lwn3\lw...
 Laboratory work № 2
Subject: Technologies of programming. Delivered on 5 week;
 Laboratory work № 3
Subject: Technologies of programming. Not delivered;
 Calculating work.
Предмет: Logical and Functional programming. Not delivered;
 Referat on the topic: What is a Logical programming?.
Subject: Technologies of programming. Not delivered;
 Referat on the topic: Security in our life.
Subject: Security. Not delivered]

Amount of completed tasks for every week ([(week, amount);...])
[(3, 2); (5, 2)]
```

Рисунок 4 – Результат роботи програми (продовження)

2. Лістинг програмного коду

Program.fs

```
open System.Collections.ObjectModel

// Learn more about F# at http://fsharp.org
// See the 'F# Tutorial' project for more help.

//task1
let countEven list = (List.filter(fun x -> x % 2 = 0) list).Length

let scalarMultiplication firstList secondList =
    List.map2 (fun x y -> x * y) firstList secondList |> List.sumBy(fun elem -> elem)

let average list =
    List.average(list)

let rec quicksort(list, asc) =
    if(list |> List.isEmpty) then
        List.Empty
    else
        let other = List.tail list
        let firstElem = List.take(1) list
        let smallerElems = quicksort ((List.filter (
                                if(asc) then
                                    fun e -> e < firstElem.Item(0)
                                else
                                    fun e -> e >= firstElem.Item(0)) other), asc)
        let largerElems =  quicksort ((List.filter (
                                if(asc) then
                                    fun e -> e >= firstElem.Item(0)
                                else
                                    fun e -> e < firstElem.Item(0)) other), asc)
        smallerElems @ firstElem @ largerElems
//task2
type Task(subject: string) =
    member x.Subject = subject
    override x.ToString() = "\nSubject: " + x.Subject.ToString()
```

```
type LaboratoryWork(subject: string, number: int) =
    inherit Task(subject)
    member x.Number = number
    override x.ToString() = "Laboratory work № " + x.Number.ToString() + "\nSubject: " +
x.Subject.ToString()

type CW(subject: string) =
    inherit Task(subject)
    override x.ToString() = "Calculating work. \nПредмет: " + x.Subject.ToString()

type Referat(subject: string, title: string) =
    inherit Task(subject)
    member x.Title = title
    override x.ToString() = "Referat on the topic: " + x.Title.ToString() + ". \nSubject:
" + x.Subject.ToString()

//пара название - номер недели
type TaskInfo(task: Task, week: int) =
    member x.Task = task
    member x.Week = week
    override x.ToString() =
        if(x.Week <> -1) then
            task.ToString() + ". Delivered on " + week.ToString() + " week"
        else
            task.ToString() + ". Not delivered"

//список пар
type TaskList() = inherit Collection<TaskInfo>()

//академический план
type AcademicPlan(tasks: TaskList) =
    member x.Tasks = tasks
    member x.addTask(task: TaskInfo) = (new TaskList()).Add(task)
    override x.ToString() = x.Tasks.ToString()
    //задания, которые необходимо сдать по указанному предмету
    member x.getByTitle(subject: string) =
        Seq.toList(x.Tasks) |> List.filter (fun x -> x.Task.Subject = subject)
    //список рефератов, которые нужно сдать
    member x.getReferats() =
        Seq.toList(x.Tasks) |> List.filter (fun x ->
x.Task.GetType().Name.Equals("Referat"))
    //список всех оставшихся заданий
    member x.getRest() = Seq.toList(x.Tasks) |> List.filter (fun x -> x.Week = -1)
    //список заданий, которые остались несданными на указанной неделе
    member x.getRestForWeek(week: int) = Seq.toList(x.Tasks) |> List.filter (fun x ->
x.Week = -1 || x.Week > week)
    //создает список, состоящий из пар,
    //первый элемент которых равен номеру недели,
    //а второй – количеству сданных на эту неделю заданий
    member x.getPlot() =
        (Seq.toList(x.Tasks)) |> List.filter (fun x -> x.Week <> -1) |> List.countBy(fun
x -> x.Week)

[<EntryPoint>]
let main argv =
    //task1
    printfn "Task1"
    let list1 = [7;9;6;9;6;9]
    let list2 = [4;7;6;9;2;1]
    let list3 = [4.9;7.5;6.3;9.2;2.6;1.0]
```

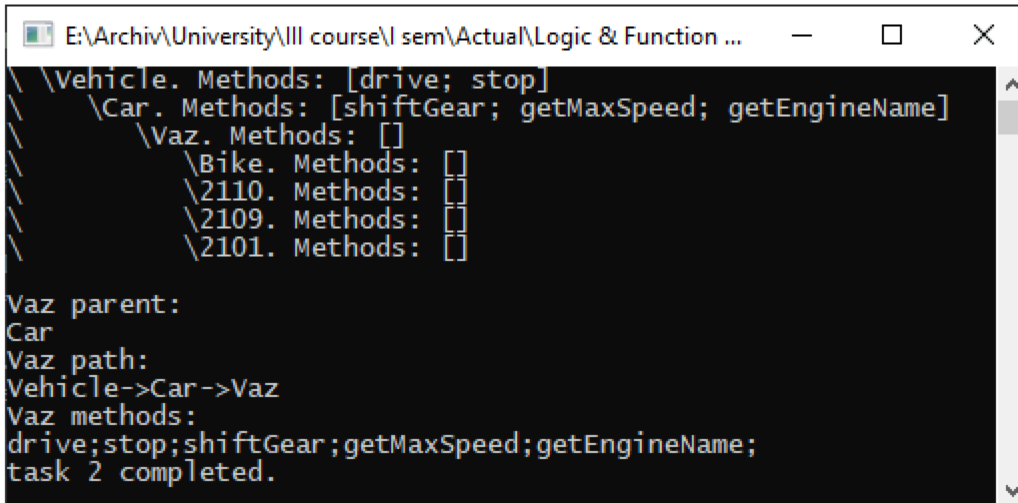
```
    let amountEven = countEven list1
    let scalarMult = scalarMultiplication list1 list2
    let avg = average list3
    let asc = false
    let sorted = quicksort (list3, asc)

    printfn "Amount even of %A: %A\n" list1 amountEven
    printfn "Scalar multiplication of %A and %A: %A\n" list1 list2 scalarMult
    printfn "Average of %A: %A\n" list3 avg
    printfn "Sorting of %A: %A\n" list3 sorted

(****************************TASK*2**************************************************)
    let tasks: TaskList = new TaskList()
    tasks.Add(new TaskInfo(new LaboratoryWork("Logical and Functional programming", 1),
3))
    tasks.Add(new TaskInfo(new LaboratoryWork("Logical and Functional programming", 2),
5))
    tasks.Add(new TaskInfo(new LaboratoryWork("Logical and Functional programming", 3), -
1))
    tasks.Add(new TaskInfo(new LaboratoryWork("Logical and Functional programming", 4), -
1))

    tasks.Add(new TaskInfo(new LaboratoryWork("Technologies of programming", 1), 3))
    tasks.Add(new TaskInfo(new LaboratoryWork("Technologies of programming", 2), 5))
    tasks.Add(new TaskInfo(new LaboratoryWork("Technologies of programming", 3), -1))
    tasks.Add(new TaskInfo(new CW("Logical and Functional programming"), -1))
    tasks.Add(new TaskInfo(new Referat("Technologies of programming", "What is a Logical
programming?"), -1))
    tasks.Add(new TaskInfo(new Referat("Security", "Security in our life"), -1))

let academPlan: AcademicPlan = new AcademicPlan(tasks)
    //задолженности
    let logFunDebt = academPlan.getByTitle("Logical and Functional programming")
    //получить список всех рефератов
    let allReferats = academPlan.getReferats()
    //список всех оставшихся заданий
    let allDebt = academPlan.getRest()
    //список заданий, которые остались несданными на указанной неделе
    let debtForWeek = academPlan.getRestForWeek(3)
    //первый элемент номеру недели, второй – количество сданных на эту неделю заданий
    let countCompletedForAllWeeks = academPlan.getPlot()

//вывод
    printfn "Task2"
    printfn "Debts for Logical and Functional programming:\n%A" logFunDebt
    printfn "\nAll referats:\n%A" allReferats
    printfn "\nAll rest tasks:\n%A" allDebt
    printfn "\nAll rest tasks for 3 week\n%A" debtForWeek
    printfn "\nAmount of completed tasks for every week ([(week, amount);...]) \n%A"
countCompletedForAllWeeks
    System.Console.ReadKey() |> ignore
    0 // return an integer exit code
```

Лабораторна робота № 4.

Варіант 20

1. Результат роботи програми

На рисунках 5 – 6 зображено результат роботи програми.



```
E:\Archiv\University\III course\I sem\Actual\Logic & Function ...
\ \Vehicle. Methods: [drive; stop]
\     \Car. Methods: [shiftGear; getMaxSpeed; getEngineName]
\         \Vaz. Methods: []
\             \Bike. Methods: []
\             \2110. Methods: []
\             \2109. Methods: []
\             \2101. Methods: []

Vaz parent:
Car
Vaz path:
Vehicle->Car->Vaz
Vaz methods:
drive;stop;shiftGear;getMaxSpeed;getEngineName;
task 2 completed.
```

Рисунок 5 – Результат роботи програми

```
matrix1.txt – Блокнот     matrix2.txt – Блокнот     matrix3.txt – Блокнот
Файл Правка Формат        Файл Правка Формат        Файл Правка Формат
2 3.9                     2 3                       4 6,9
3.2 1                     2 5                       5,2 6
2 4                       2 2                       4 6
100% Windows (CRLF)       100% Windows (CRLF)       100% UNIX (LF)
```

Рисунок 6 – Файли matrix1-3

2. Лістинг програмного коду

Program.fs

```
open System.IO
open System

//////////////////TASK/1/////////////////////

type Class(name: string, methods:string list) =
    member x.Name = name
    member x.Methods = methods
    override x.ToString() =
        x.Name + ". Methods: " + x.Methods.ToString()

type 'T tree =
    Leaf of 'T
    | Node of 'T*('T tree list)

let iterh f =
    let rec itr n = function
```

```
    Leaf(T) -> f n T
    | Node(T,L) -> (f n T;for t in L do itr (n+1) t done) in itr 0

let rec iter f = function
    Leaf(T) -> f T
    | Node(T,L) -> (f T; for t in L do iter f t done)

let spaces n = List.fold (fun s _ -> s+" ") "" [0..n]
let print_tree T = iterh (fun h x -> printfn "\%s\%s " (spaces (h*3))
(x.ToString())) T

let rec getParent name lst tree =
    match tree with
    | Node(n:Class, sub) -> if (n.Name = name) then
                                if (lst |> List.length) <> 0 then lst
                                else ["null"] //если предка нет
                            else List.collect (getParent name (n.Name::[])) sub
    | Leaf(n:Class) -> if (n.Name = name) then lst else []


let rec getPath name lst tree =
    match tree with
    | Node(n:Class, sub) -> if (n.Name = name) then List.rev (n.Name::lst) else
List.collect (getPath name (n.Name::lst)) sub
    | Leaf(n:Class) -> if (n.Name = name) then List.rev (n.Name::lst) else []


let rec inh x name (y:Class) =
    match x with                                       //стрелка вниз пройтись по
всем элементам
    | Node(L:Class,T) when L.Name <> name -> Node(L, [inh T.Head name y])
    | Node(L:Class,T) when L.Name = name -> Node(L, [Leaf(y)]@T)
    | Leaf(T:Class) -> if (T.Name = name) then Node(T, [Leaf(y)]) else Leaf(T)

   let rec getMethods name lst tree =
    match tree with
    | Node(n:Class, sub) -> if (n.Name = name) then List.rev (n.Methods::lst)
                            else List.collect (getMethods name (n.Methods::lst))
sub
    | Leaf(n:Class) -> if (n.Name = name) then List.rev (n.Methods::lst)
                       else []

let showList list delim =
    list |> List.iteri (fun i x -> if i = 0 then printf("%s") x else
printf("%s%s") delim x)

let showMethods list =
    list |> List.iteri (fun i x -> if i <> (List.length list - 1) then showList x
";"; printf(";") else showList x ";")

//////////////////////////TASK/2////////////////////////////////
let readLines filePath = System.IO.File.ReadAllLines(filePath)
let m1 = readLines "C:\Users\Methew Snipes\Desktop\matrix1.txt"
let m2 = readLines "C:\Users\Methew Snipes\Desktop\matrix2.txt"

let writeLines content = System.IO.File.AppendAllText("C:\Users\Methew
Snipes\Desktop\matrix3.txt", content)

let strToArray (pattern: Char) (text: string) =
    let words = text.Split [|pattern|]
    (words)
```

```
let funct (arr: array<string>) =
    let mutable list = new Collections.Generic.List<string[]>()
    for c in arr do
        let tmp = strToArray ' ' (c)
        list.Add(tmp) |> ignore
    list

let sumVecs (v1:string[], v2:string[]) =
    Array.map2 (fun x y -> (float(x) + float(y)).ToString()) v1 v2

let sumMatr (matr1:Collections.Generic.List<string[]>)
(matr2:Collections.Generic.List<string[]>) =
    let mutable list = new Collections.Generic.List<string[]>()

    for i in 0..matr1.Count - 1 do
        list.Add(sumVecs(matr1.Item(i),matr2.Item(i)))
    list

let writeMatrToFile (matr: Collections.Generic.List<string[]>) =
    for el in matr do
        for c in el do
            writeLines c
            writeLines " "
        writeLines "\n"

[<EntryPoint>]
let main argv =
    (*************TASK*1****************)
    let str = "Vaz"
    let tra = Node(new Class("Vehicle", ["drive"; "stop"]) , [Node(new
Class("Car", ["shiftGear";"getMaxSpeed"; "getEngineName"]), [Node(new Class("Vaz",
[]), [Leaf(new Class("2110", [])); Leaf(new Class("2109", [])); Leaf(new
Class("2101", []))])]); Node(new Class("Truck", ["getCarryingCapacity"]),
[Node(new Class("Kamaz", ["transportingCargo"]), [Leaf(new
Class("51153",[]))])])])
    let newClass = "Bike";

    let n = new Class(newClass, [])
    let a = (inh tra str n)

    print_tree a

    printfn "\n%s parent:" str
    showList (getParent str [] tra) ""
    printfn "\n%s path: " str
    showList (getPath str [] tra) "->"
    printfn "\n%s methods: " str
    showMethods (getMethods str [] tra)


    (*************TASK*2****************)
    //printfn "m1=%A\n" (funct m1)
    //printfn "m2=%A" (funct m2)
    let matr1 = (funct(m1))
    let matr2 = (funct(m2))
    let sum = sumMatr matr1 matr2
    //printfn "m2=%A" (sum)
    writeMatrToFile sum //|> ignore
    printfn "\ntask 2 completed."

    System.Console.ReadKey() |> ignore
    0 // return an integer exit code
```

ДОДАТОК А
ТЕОРЕТИЧНИЙ МАТЕРІАЛ
ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 1

1. Основи роботи з інтерпретатором Hugs

Для виконання лабораторних робіт буде використовуватися інтерпретатор мови Haskell. Існує кілька реалізацій інтерпретатора; в цьому курсі буде використовуватися інтерпретатор Hugs (ця назва є абревіатурою слів «Haskell users 'Gofer system», Gofer - назва мови програмування, який був одним з попередників Haskell.)

Після запуску інтерпретатора Hugs на екрані з'являється діалогове вікно середовища розробника, автоматично завантажується спеціальний файл визначених типів і визначень стандартних функцій на мові Haskell (Prelude.hs), і виводиться стандартне запрошення до роботи. Це запрошення має вигляд Hugs>; взагалі, перед символом> виводиться ім'я останнього завантаженого модуля.

Після виведення запрошення можна вводити вирази мови Haskell, або команди інтерпретатора. Команди інтерпретатора відрізняються від виразів мови Haskell тим, що починаються з символу двокрапки (:). Прикладом команди інтерпретатора є команда: quit, по якій відбувається завершення роботи інтерпретатора. Команди інтерпретатора можна скорочувати до однієї літери; таким чином, команди: quit і: q еквівалентні. Команда: set використовується для того, щоб встановити різні опції інтерпретатора. Команда:? виводить список доступних команд інтерпретатора. Надалі ми розглянемо інші команди.

2. Типи

Програми на мові Haskell є вираження, обчислення яких приводить до значень. Кожне значення має тип. Інтуїтивно тип можна розуміти просто як безліч допустимих значень виразу. Для того, щоб дізнатися тип деякого виразу, можна використовувати команду інтерпретатора: type (або: t). Крім того, можна виконати команду: set + t, для того, щоб інтерпретатор автоматично друкував тип кожного обчисленого результату.

Основними типами мови Haskell є:

• Типи Integer і Int використовується для представлення цілих чисел, причому значення типу Integer не обмежені по довжині.

• Типи Float і Double використовується для подання дійсних чисел.

• Тип Bool містить два значення: True і False, і призначений для представлення результату логічних виразів.

• Тип Char використовується для представлення символів.

Імена типів в мові Haskell завжди починаються з великої літери.

Мова Haskell є сильно універсальна мова програмування. Проте в більшості випадків програміст не зобов'язаний оголошувати, яким типам

належать вводяться ним змінні. Інтерпретатор сам здатний вивести типи вживаних користувачем змінних. Однак, якщо все ж для будь-яких цілей необхідно оголосити, що деяке значення належить деякому типу, використовується конструкція виду: змінна:: Тип. Якщо включена опція інтерпретатора + t, він друкує значення в такому ж форматі.

Нижче наведено приклад протоколу сесії роботи з інтерпретатором. Передбачається, що текст, наступний за запрошенням Hugs », вводить користувач, а наступний за цим текст являє відповідь системи.

```
Hugs>: set + t
Hugs> 1
1 :: Integer
Hugs> 1.2
1.2 :: Double
Hugs> 'a'
'A':: Char
Hugs> True True :: Bool
```

З даного протоколу можна зробити висновок, що значення типу Integer, Double і Char задаються за тими ж правилами, що і в мові Сі.

Розвинена система типів і сувора типізація роблять програми на мові Haskell безпечними за типами. Гарантується, що в правильній програмі на мові Haskell всі типи використовуються правильно. З практичної точки зору це означає, що програма на мові Haskell при виконанні не може викликати помилок доступу до пам'яті (Access violation). Також гарантується, що в програмі не може статися використання неініціалізованих змінних. Таким чином, багато помилок в програмі відстежуються на етапі її компіляції, а не виконання.

3. Арифметика

Інтерпретатор Hugs можна використовувати для обчислення арифметичних виразів. При цьому можна використовувати оператори +, -, *, / (додавання, віднімання, множення і ділення) зі звичайними правилами пріоритету. Крім того, можна використовувати оператор ^ (піднесення до степеня). Таким чином, сеанс роботи може виглядати наступним чином:

```
Hugs> 2 * 2
4 :: Integer
Hugs> 4 * 5 + 1
21 :: Integer
Hugs> 2 ^ 3
8 :: Integer
```

Крім того, можна використовувати стандартні математичні функції sqrt (квадратний корінь), sin, cos, exp і т.д. На відміну від багатьох інших мов програмування, в Haskell при виконанні функції не обов'язково поміщати аргумент в дужки. Таким чином, можна просто писати sqrt 2, а не sqrt (2).

Приклад:

```
Hugs> sqrt 2
1.4142135623731 :: Double
Hugs> 1 + sqrt 2
2.4142135623731 :: Double
Hugs> sqrt 2 + 1
2.4142135623731 :: Double
Hugs> sqrt (2 + 1)
1.73205080756888 :: Double
```

З цього прикладу можна зробити висновок, що виклик функції має більш високий пріоритет, ніж арифметичні операції, так що вираз sqrt 2 + 1 інтерпретується як (sqrt 2) + 1, а не sqrt (2 + 1). Для завдання точного порядку обчислення слід використовувати дужки, як в останньому прикладі. (Насправді виклик функції має більш високий пріоритет, ніж будь-який бінарний оператор.)

Також слід зауважити, що на відміну від більшості інших мов програмування, цілочисельні вирази в мові Haskell обчислюються з необмеженим числом розрядів (Спробуйте обчислити вираз 2 ^ 5000.) На відміну від мови Сі, де максимально можливе значення типу int обмежена розрядністю машини (на сучасних персональних комп'ютерах воно дорівнює 231 - 1 = 2147483647), тип Integer в мові Haskell може зберігати цілі числа довільної довжини.

4. Кортежі

Крім перерахованих вище простих типів, в мові Haskell можна визначати значення складових типів. Наприклад, для завдання точки на площині необхідні два числа, що відповідають її координатами. У мові Haskell пару можна задати, перерахувавши компоненти через кому і взявши їх в дужки: (5,3). Компоненти пари не обов'язково повинні належати одному типу: можна скласти пару, першим елементом якої буде рядок, а другим - число і т.д.

У загальному випадку, якщо a і b - деякі довільні типи мови Haskell, тип пари, в якій перший елемент належить типу a, а другий - типу b, позначається як (a, b). Наприклад, пара (5,3) має тип (Integer, Integer); пара (1, 'a') належить типу (Integer, Char). Можна навести й більш складний приклад: пара ((1, 'a'), 1.2) належить типу ((Integer, Char), Double). Перевірте це за допомогою інтерпретатора.

Слід звернути увагу, що хоча конструкції виду (1,2) і (Integer, Integer) виглядають схоже, в мові Haskell вони позначають зовсім різні сутності. Перша є значенням, в той час як остання - типом.

Для роботи з парами в мові Haskell існують стандартні функції fst і snd, які повертають, відповідно, перший і другий елементи пари (назви цих

функцій відбуваються від англійських слів «first» (перший) і «second» (другий)). Таким чином, їх можна використовувати в такий спосіб

```
Hugs> fst (5, True)
5 :: Integer
Hugs> snd (5, True)
True :: Bool
```

Крім пар, аналогічним чином можна визначати трійки, четвірки і т.д. Їх типи записуються аналогічним чином.

```
Hugs> (1,2,3)
(1,2,3) :: (Integer, Integer, Integer)
Hugs> (1,2,3,4)
(1,2,3,4) :: (Integer, Integer, Integer, Integer)
```

Така структура даних називається кортежем. У кортежі може зберігається фіксована кількість різнорідних даних. Функції fst і snd визначені тільки для пар і не працюють для інших кортежів. При спробі використовувати їх, наприклад, для трійок, інтерпретатор видає повідомлення про помилку.

Елементом кортежу може бути значення будь-якого типу, в тому числі і інший кортеж. Для доступу до елементів кортежів, складених з пар, може використовуватися комбінація функцій fst і snd.

Наступний приклад демонструє витяг елемента 'a' з кортежу (1, ('a', 23.12)):

```
Hugs> fst (snd (1, ( 'a', 23.12)))
'A' :: Integer
```

5. Списки

На відміну від кортежів, список може зберігати будь-яку кількість елементів. Щоб задати список в Haskell, необхідно в квадратних дужках перерахувати його елементи через кому. Всі ці елементи повинні належати одному і тому ж типу. Тип списку з елементами, що належать типу a, позначається як [a].

```
Hugs> [1,2]
[1,2] :: [Integer]
Hugs> [ '1', '2', '3']
[ '1', '2', '3'] :: [Char]
```

У списку може не бути жодного елемента. Порожній список позначається як [].

Оператор: (двокрапка) використовується для додавання елемента в початок списку. Його лівим аргументом повинен бути елемент, а правим – список:

```
Hugs> 1: [2,3]
[1,2,3] :: [Integer]
Hugs> '5': [ '1', '2', '3', '4', '5']
[ '5', '1', '2', '3', '4', '5'] :: [Char]
Hugs> False: []
[False] :: [Bool]
```

За допомогою оператора (:) і порожнього списку можна побудувати будь-який список:

```
Hugs> 1: (2: (3: []))
[1,2,3] :: Integer
```

Оператор (:) асоціативний вправо, тому в наведеному вище виразі можна опустити дужки:

```
Hugs> 1: 2: 3: []
[1,2,3] :: Integer
```

Елементами списку можуть бути будь-які значення - числа, символи, кортежі, інші списки і т.д.

```
Hugs> [(1, 'a'), (2, 'b')]
[(1, 'a'), (2, 'b')] :: [(Integer, Char)]
Hugs> [[1,2], [3,4,5]]
[[1,2], [3,4,5]] :: [[Integer]]
```

Для роботи зі списками в мові Haskell існує велика кількість функцій. У даній лабораторній роботі розглянемо тільки деякі з них.

- Функція head повертає перший елемент списку.
- Функція tail повертає список без першого елемента.
- Функція length повертає довжину списку.

Функції head і tail визначені для непустих списків. При спробі застосувати їх до порожнього списку інтерпретатор повідомляє про помилку.

Приклади роботи з зазначеними функціями:

```
Hugs> head [1,2,3]
1 :: Integer
Hugs> tail [1,2,3]
[2,3] :: [Integer]
Hugs> tail [1]
[] :: Integer
Hugs> length [1,2,3]
3 :: Int
```

Зауважте, що результат функції length належить типу Int, а не типу Integer.

Для з'єднання (конкатенації) списків в Haskell визначений оператор ++.

```
Hugs> [1,2] ++ [3,4]
[1,2,3,4] :: Integer
```

6. Рядки

Строкові значення в мові Haskell, як і в Сі, задаються в подвійних лапках. Вони належать типу String.

```
Hugs> "hello"
"Hello" :: String
```

Насправді рядки є списками символів; таким чином, вирази "hello", ['h', 'e', 'l', 'l', 'o'] і 'h': 'e': 'l': 'l': 'o': [] означають одне і те ж, а тип String є синонімом для [Char]. Всі функції для роботи зі списками можна використовувати при роботі з рядками:

```
Hugs> head "hello"
'H' :: Char
Hugs> tail "hello"
"Ello" :: [Char]
Hugs> length "hello"
5 :: Int
Hugs> "hello" ++ ", world"
"Hello, world" :: [Char]
```

Для перетворення числових значень в рядки і навпаки існують функції read і show:

```
Hugs> show 1
"1" :: [Char]
Hugs> "Formula" ++ show 1
"Formula 1" :: [Char]
Hugs> 1 + read "12"
13 :: Integer
```

Якщо функція show не зможе перетворити рядок в число, вона повідомить про помилку.

7. Функції

До сих пір ми використовували вбудовані функції мови Haskell. Тепер прийшла пора навчитися визначати власні функції. Для цього нам необхідно вивчити ще кілька команд інтерпретатора (нагадаємо, що ці команди можуть бути скорочені до однієї літери):

• Команда: load дозволяє завантажити в інтерпретатор програму на мові Haskell, що міститься в зазначеному файлі.

• Команда: edit запускає процес редагування останнього завантаженого файлу.

• Команда: reload перечитує останній долучення.

Визначення функцій користувача повинні перебувати в файлі, який потрібно завантажити в інтерпретатор Hugs за допомогою команди: load. Для редагування завантаженої програми можна використовувати команду: edit. Вона запускає зовнішній редактор (за замовчуванням це Notepad) для редагування файлу. Після завершення сеансу редагування редактор необхідно закрити; при цьому інтерпретатор Hugs перечитає вміст зміненого файлу. Однак файл можна редагувати і безпосередньо з оболонки Windows. В цьому випадку, для того щоб інтерпретатор зміг перечитати файл, необхідно явно викликати команду: reload.

Розглянемо приклад.

Створіть в какомлибо каталозі файл lab1.hs. Нехай повний шлях до цього файлу - з: \ labs \ lab1.hs (це тільки приклад, ваші файли можуть називатися по іншому). У інтерпретаторі Hugs виконайте наступні команди:

```
Hugs>: load "з: \\ labs \\ labl.hs"
```

Якщо завантаження проведена успішно, запрошення інтерпретатора змінюється на Main>. Справа в тому, що якщо не вказано ім'я модуля, вважається, що воно дорівнює Main.

```
Main>: edit
```

Тут має відкритися вікно редактора, в якому можна вводити текст програми. Введіть:

```
x = [1,2,3]
```

Збережіть файл і закрийте редактор. Інтерпретатор Hugs завантажить пакунки: \ labs \ lab1.hs і тепер значення змінної x буде визначено:

```
Main> x [1,2,3] :: [Integer]
```

Зверніть увагу, що під час запису імені файлу в аргументі команди: load символи \ дублюються. Також, як і в мові Ci, в Haskell символ \ служить індикатором початок службового символу ('\ n' і т.п.) Для того, щоб ввести безпосередньо символ \, необхідно, як і в Ci, екранувати його ще одним символом \ .

Тепер можна перейти до визначення функцій. Створіть, в відповідність з процесом, описаним вище, какойлібо файл і запишіть в нього наступний текст:

```
square :: Integer -> Integer
square x = x * x
```

Перший рядок (square:: Integer> Integer) оголошує, що ми визначаємо функцію square, приймаючу параметр типу Integer і повертає результат типу Integer. Другий рядок (square x = x * x) є безпосередньо визначенням функції. Функція square приймає один аргумент і повертає його квадрат.

Функції в мові Haskell є значеннями «першого класу». Це означає, що вони «рівноправні» з такими значеннями, як цілі і речові числа, символи, рядки, списки і т.д. Функції можна передавати в якості аргументів в інші функції, повертати їх з функцій і т.п. Як і всі значення в мові Haskell, функції мають тип. Тип функції, що приймає значення типу a і повертає значення типу b позначається як a> Ь.

Завантажте створений файл в інтерпретатор і виконайте наступні команди:

```
Main>: type square
square :: Integer -> Integer
Main> square 2
4 :: Integer
```

Зауважимо, що в принципі оголошення типу функції square не була необхідною: інтерпретатор сам міг вивести необхідну інформацію про тип функції з її визначення. Однак, по-перше, виведений тип був би більш загальним, ніж Integer> Integer, а по-друге, явна вказівка типу функції є «хорошим тоном» при програмуванні на мові Haskell, оскільки оголошення типу служить свого роду документацією до функції і допомагає виявляти помилки програмування.

Імена визначаються користувачем функцій і змінних повинні починатися з літери в нижньому регістрі. Решта символів в імені можуть бути великими або малими латинськими буквами, цифрами або символами _ і '(підкреслення і апостроф). Таким чином, нижче перераховані приклади правильних імен змінних:

```
var
var1
variableName
variable_name
var '
```

8. Умовні вирази

У визначенні функції в мові Haskell можна використовувати умовні вирази. Запишемо функцію signum, яка обчислює знак переданого їй аргументи:

```
signum :: Integer -> Integer
signum x = if x> 0 then 1
 else if x <0 then -1
else 0
```

Умовний вираз записується у вигляді:

if умова then вираз else вираз.

Зверніть увагу, що хоча з вигляду цей вислів нагадує відповідний оператор в мові Сі або Паскаль, в умовному вираженні мови Haskell повинні бути присутніми і thenчасть і elseчасть. Вирази в thenчасті і в elseчасті умовного оператора повинні належати одному типу.

Умова у визначенні умовного оператора є будь-який вираз типу Bool. Прикладом таких виразів можуть служити порівняння. При порівнянні можна використовувати наступні оператори:

- <,>, <=,> = - ці оператори мають такий же зміст, як і в мові Сі (менше, більше, менше або дорівнює, більше або дорівнює).
- == - оператор перевірки на рівність.
- / = - оператор перевірки на нерівність.

Вирази типу Bool можна комбінувати за допомогою загальноприйнятих логічних операторів && і | | (І і АБО), і функції заперечення not.

Приклади допустимих умов:

```
x> = 0 && x <= 10
x > 3 && x / = 10
(X> 10 || x <10) && not (x == y)
```

Зрозуміло, можна визначати свої функції, які повертають значення типу Bool, і використовувати їх в якості умов. Наприклад, можна визначити функцію isPositive, що повертає True, якщо її аргумент неотрицателен і False в іншому випадку:

```
isPositive :: Integer -> Bool
isPositive x = if x> 0 then True else False
```

Тепер функцію signum можна визначити наступним чином:

```
signum :: Integer -> Integer
signum x = if isPositive x then 1
 else if x <0 then -1
else 0
```

Відзначимо, що функцію isPositive можна визначити і простіше:

```
isPositive x = x> 0
```

9. Функції багатьох змінних і порядок визначення функцій

До сих пір ми визначали функції, які беруть один аргумент. Зрозуміло, в мові Haskell можна визначати функції, які беруть довільну кількість аргументів. Визначення функції add, що приймає два цілих числа і повертає їх суму, виглядає наступним чином:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

Тип функції add може виглядати дещо загадково. У мові Haskell вважається, що операція> асоціативна вправо. Таким чином, тип функції add може бути прочитаний як Integer> (Integer> Integer), тобто у відповідність з правилом каррінг, результатом застосування функції add до одного аргументу буде функція, приймаюча один параметр типу Integer. Взагалі, тип функції, що приймає n аргументів, що належать типам t1, t2, ..., tn, і повертає результат типу a, записується у вигляді t1> t2> ...> tn> a

Слід зробити ще одне зауваження, що стосується порядку визначення функцій. У попередньому розділі ми визначили дві функції, signum і isPositive, одна з яких використовувала для свого визначення іншу. Виникає питання: яка з цих функцій повинна бути визначена раніше? Напрошується відповідь, що визначення isPositive має передувати визначення функції signum; проте в дійсності в мові Haskell порядок визначення функцій не має значення! Таким чином, функція isPositive може бути визначена як до, так і після функції signum.

ДОДАТОК Б
ТЕОРЕТИЧНИЙ МАТЕРІАЛ
ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 2

1. Коментарі

Необхідність наявності коментарів в програмі очевидна. На жаль, автори різних мов програмування розходяться між собою в питанні про те, яким чином позначати коментарі в коді. Haskell не став винятком.

У мові Haskell, як і в C ++, визначені два види коментарів: рядкові і блокові. Рядковий коментар починається з символів - і триває до кінця рядка (аналогом в C ++ служить коментар, що починається з //). Блоковий коментар починається символами {-І триває до символів} (аналог в C ++ - коментар, обмежений символами / * І * /. Зрозуміло, все, що є коментарем, ігнорується інтерпретатором або компілятором мови Haskell. **Приклад:**

```
fx = x -Це коментар
g x y = {-Це теж коментар. Тільки довше.} X + y
```

2. Рекурсія

В імперативних мовах програмування основною конструкцією є цикл. У Haskell замість циклів використовується рекурсія. Функція називається рекурсивної, якщо вона викликає сама себе (або, точніше, визначена в термінах самої себе). Рекурсивні функції існують в імперативних мовах, але використовуються не настільки широко. Однією з найпростіших рекурсивних функцій є факторіал:

```
factorial :: Integer -> Integer
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Зауважте, що ми пишемо factorial (n-1), а не factorial n-1 – згадайте про пріоритети операцій.

Використання рекурсії може викликати труднощі. Концепція рекурсії нагадує про застосовуватися в математиці прийомі докази по індукції. У нашому визначенні факторіала ми виділяємо «базу індукції» (випадок n == 0) і «крок індукції» (перехід від factorial n до factorial (n-1). Виділення таких компонент - важливий крок у визначенні рекурсивної функції.

3. Операція вибору і правила вирівнювання

Раніше було розглянуто умовний оператор. Його природним продовженням є оператор вибору case, аналогічний конструкції switch мови Сі.

Припустимо, нам треба визначити деяку (досить дивну) функцію, яка повертає 1, якщо їй переданий аргумент 0; 5, якщо аргумент дорівнював 1;

2, якщо аргумент дорівнює 2 і 1 у всіх інших випадках. В принципі, цю функцію можна записати за допомогою операторів if, проте результат буде довгим і малозрозумілим. У таких випадках допомагає використання case:

```
fx = case x of
0 ->1
1 ->5
2 ->2
_ -> 1
```

Синтаксис оператора case очевидний з наведеного прикладу; слід тільки зауважити, що символ _ аналогічний конструкції default в мові Сі. Однак у уважного читача може виникнути закономірне питання: яким чином інтерпретатор мови Haskell розпізнає, де закінчилося визначення одного випадку і почалося визначення іншого?

Відповідь полягає в тому, що в мові Haskell використовується двовимірна система структурування тексту (аналогічна система використовується в більш широко відомою мовою Python). Ця система дозволяє обійтися без спеціальних символів угруповання і поділу операторів, подібним символам {,} і; мови Сі.

Насправді в мові Haskell також можна використовувати ці символи в тому ж сенсі (За тим винятком, що в Haskell, як і в мові Паскаль, символ ';' використовується як роздільник операторів, а не як ознака завершення оператора.). Так, вищенаведену функцію можна записати і таким чином (що демонструє, як не потрібно оформляти тексти програм):

```
fx = case x of
{0 -> 1; 1 -> 5;
 2 -> 2;
 __> 1}
```

Такий спосіб явно задає угруповання і поділ конструкцій мови. Однак можна обійтися і без нього.

Загальне правило таке. Після ключових слів where, let, do і of інтерпретатор вставляє відкриває дужку ({) і запам'ятовує колонку, в якій записана наступна команда. Надалі перед кожною новою рядком, вирівняною на запомненную величину, вставляється розділяє символ ';'. Якщо такий рядок вирівняна менше (тобто її перший символ знаходиться лівіше позиції, яка запам'ятається), вставляється закриває дужка. Це може виглядати дещо важкувато, але в дійсності все досить просто.

Застосовуючи описане правило до визначення функції f, отримаємо, що воно сприймається інтерпретатором наступним чином:

```
fx = case x of {
; 0 -> 1
; 1 -> 5
```

```
; 2 -> 2
; -> 1
}
```

У будь-якому випадку можна не використовувати цей механізм і завжди явно вказувати символи {,} і;. Однак, крім економії на кількості натискань клавіш, застосування описаного правила призводить до того, що отримуються програми більш «читабельні». Таким чином, для лабораторних робіт пропонується зробити вживання такого оформлення обов'язковим.

Необхідно зробити ще одне зауваження. Оскільки в програмі на мові Haskell прогалини є значущими, необхідно бути уважними

до використання символів табуляції. Інтерпретатор вважає, що символ табуляції дорівнює 8 прогалин. Однак деякі текстові редактори дозволяють налаштовувати відображення табуляції і робити його еквівалентним іншому числу прогалин (наприклад, за замовчуванням в редакторі Visual Studio табуляція відображається як 4 пробілу). Це може привести до помилок, якщо поєднувати в одній програмі прогалини і табуляцію. Найкраще при програмуванні на Haskell взагалі не використовувати табуляцію (багато редакторів дозволяють вводити після натискання клавіші табуляції вказане число пробілів).

4. Кускове завдання функцій

Функції можуть бути визначені КУСКОВО чином (згадайте поняття кусочнопостоянних або кусочнолінейних функцій в математиці). Це означає, що можна визначити одну версію функції для певних параметрів і іншу версію для інших параметрів. Так, функцію f з попереднього розділу можна визначити наступним чином:

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

Порядок визначення в даному випадку важливий. Якби ми записали спершу визначення f _ = -1, то f повертала б -1 для будь-якого аргументу. Якби ми зовсім не вказали цей рядок, ми отримали б помилку, якби спробували вирахувати її значення для аргументу, відмінного від 0, 1 або 2.

Такий спосіб визначення функцій досить широко використовується в мові Haskell. Він часто дозволяє обійтися без операторів if і case. Так, функцію факторіала можна визначити в такому стилі:

```
factorial 0 = 1
factorial n = n * factorial (n -1)
```

5. Зіставлення зі зразком

Крім рекурсивних функцій на цілих числах, можна визначати рекурсивні функції на списках. У цьому випадку «базою рекурсії» буде порожній список ([]). Визначимо функцію обчислення довжини списку (оскільки ім'я length вже зайнято стандартної бібліотекою, назвемо її len):

```
len [] = 0
len s = 1 + len (tail s)
```

Згадаймо, що список, першим елементом якого (головою) є x, а інші елементи (хвіст) задаються списком xs, записується як x: xs. Виявляється, подібну конструкцію можна застосовувати при описі функції:

```
len []  = 0
len (x: xs) = 1 + len xs
```

Рядок xs слід розуміти, як множина від x, утворене за правилами англійської мови.

Наведемо ще один приклад. Функцію, приймаючу на вхід пару чисел і повертає їх суму, можна визначити таким чином:

```
sum_pair p = fst p + snd p
```

Однак що робити, якщо необхідно визначити функцію, приймаючу трійку чисел і повертає їх суму? У нашому розпорядженні немає функцій, подібних fst і snd, для вилучення елементів трійки. Виявляється, можна записувати такі функції наступним чином:

```
sum_pair (x, y) = x + y
sum_triple (x, y, z) = x + y + z
```

Такий прийом називається зіставлення зі зразком (Від англійського «pattern matching»). Він є дуже потужною конструкцією мови, яка використовується у багатьох його місцях, зокрема, в аргументах функцій і в варіантах оператора case. «Зразки», що записуються в аргументах функції, «зіставляються» з переданими в неї фактичними параметрами.

Якщо відбувається зіставлення зі зразком, згадані в ньому змінні отримують відповідні значення. Якщо ці значення не потрібні при обчисленні функції (як у функції my_tail в наступному прикладі), то, щоб не вводити зайвих імен, можна використовувати символ _. Він означає зразок, з яким може зіставити будь-яке значення, але саме це значення не зв'язується ні з якої змінної.

Наступні приклади показують різні варіанти застосування зіставлення зі зразком:

```
--Функція підсумовування двох перших елементів списку
f1 (x: y: xs) = x + y
-Визначення функції, аналогічної head
my_head (x: xs) = x
--Визначення функції, аналогічної tail.
--Ми використовуємо _, оскільки нам не потрібно значення
--первого елемента списку
my_tail (_: xs) = xs
--Функція вилучення першого елемента трійки
fst3 (x, _, _) = x
Зіставлення зі зразком можна застосовувати і в операторі case:
--Еще одне визначення функції довжини списку my_length
 s = case s of
[] -> 0
(_: Xs) -> 1 + my_length xs
```

Можна ставити досить складні зразки. Визначимо функцію, приймаючу список пар чисел і повертає суму їх різниць (тобто f [(x1, y1), (x2, y2), ..., (xn, yn)] = (X1 -y1) + (x2 y2) + ... + (xn -yn)):

```
f [] = 0
f ((x, y): xs) = (x -y) + f xs
```

6. Побудова списків

При визначенні функцій, які повертають список, часто використовується оператор:. Наприклад, функція, що приймає список чисел і повертає список їх квадратів, може бути визначена наступним чином:

```
square [] = []
square (x: xs) = x * x: square xs
```

7. Деякі корисні функції

При виконанні лабораторної роботи можуть знадобитися такі стандартні функції мови Haskell:

even – повертає True для парного аргументу і False для непарного.

odd – аналогічно попередньої, але аргумент перевіряється на непарність.

ДОДАТОК В
ТЕОРЕТИЧНИЙ МАТЕРІАЛ
ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 3

Призначені для користувача типи даних в мові Haskell. Функції вищого порядку.

1. let-зв'язування

При визначенні функцій часто буває необхідно використовувати деякі тимчасові змінні для зберігання проміжних результатів. Згадаймо, як обчислюються коріння квадратного рівняння виду ax2 + b x + c = 0: x1,2 = (-b $\pm\sqrt{b^2-4ac}$) / 2a. Можна записати наступну функцію для обчислення пари коренів рівняння:

```
roots ab з =
((-B + sqrt (b * b-4 * a * c)) / (2 * a),
(- b-sqrt (b * b-4 * a * c)) / (2 * a))
```

Написання функцій в такому стилі може спричинити проблеми. По-перше, легко припуститися помилки, другий раз записуючи одне і те ж вираз. По-друге, при читанні цієї програми доводиться зіставляти два вирази, щоб зрозуміти, що вони представляють собою одне й те саме. По-третє, програма стає довшим. І нарешті, вона менш ефективна, ніж могла б бути, тому що комп'ютера доводиться два рази проводити однакові обчислення.

Для уникнення цих проблем в мові можна вводити локальні змінні. Функцію можна записати так:

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

Локальна змінна det доступна тільки у визначенні функції roots.Можна визначати декілька локальних змінних:

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

Зауважте, що в конструкції let ... in ... використовується правило вирівнювання: перший символ, відмінний від пропуску, наступний за ключовим словом let, задає колонку, щодо якої повинні вирівнюватися наступні визначення. З використанням символів '{', '}' і ';' правило

вирівнювання стає необов'язковим, і функцію roots можна було б записати так:

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

Крім конструкції let ... in ... іноді зручніше використовувати конструкцію. . . where ..., в якій визначення локальних змінних слідують після основної функції:

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

Зауважимо, що в принципі можна було не вводити локальних змінних і використовувати замість них глобальні функції:

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

Однак недоліки такого підходу очевидні: крім того неприємного факту, що ми ввели дві допоміжні функції в глобальному просторі імен (а це значить, що ми тепер не зможемо використовувати, наприклад, ім'я det для будь-якої іншої корисної функції), для обчислення значень $\sqrt{b^2 - 4ac}$ і 2a доводиться передавати відповідні параметри в функції, тоді як локальні визначення можуть вільно використовувати параметри функції, в рамках якої вони визначені.

У конструкціях let і where можна визначати не тільки змінні, але і функції. Розглянемо, наприклад, функцію, що повертає по заданому числу n список натуральних чисел [1, 2, ..., n]. Введемо допоміжну функцію numsFrom, яка за заданою кількістю m повертає список [m, m + l, m + 2, ..., n] і зробимо його визначення локальним:

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

Зауважте, що функція numsFrom використовує в своїй ухвалі змінну n, хоча вона не передається в неї в якості параметра.

2. Сигналізація про помилки

Обумовлені нами функції можуть не обчислюватися при деяких значеннях аргументу. Згадаймо визначення функції факторіалу:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Ця функція чудово працює до тих пір, поки ми не спробуємо вирахувати факторіал негативного числа. Неважко помітити, що в цьому випадку обчислення йде в нескінченну рекурсію, оскільки базовий випадок ніколи не досягається.

Найпростіший спосіб сигналізувати про таких помилках - використовувати стандартну функцію error. Вона приймає в якості аргументу рядок, а її обчислення призводить до зупинки програми і видачі на екран цього рядка. Таким чином, функцію факторіала запишемо так:

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

3. Охороняють умови

Зіставлення зі зразком надає широкі можливості у визначенні функцій. Однак з його допомогою, по суті, можна виділити тільки структуру переданих у функцію параметрів і рівність елементів цих параметрів сталою значенням. Однак найчастіше цього недостатньо: необхідно накладати більш складні умови на вхідні параметри.

Наприклад, у наведеному вище визначенні функції factorial ми використовували поєднання зіставлення зі зразком і умовного оператора. Зіставлення зі зразком виглядає економніше і наочніше. Чи можна використовувати схожий синтаксис для умов? Так, при використанні охороняють умов. З ними функція факторіала запишеться наступним чином:

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

Синтаксис очевидний з наведеного прикладу. Також зауважимо, що замість останнього умови можна використовувати слово otherwise (англ. Інакше, в іншому випадку). Наприклад, функція визначення знака числа виглядає наступним чином:

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

Визначення функцій в такому стилі зазвичай наочніше і в програмах на Haskell охороняють умови використовуються дуже широко (відповідно, умовний оператор використовується рідко). Для ілюстрації наочності наведемо визначення функції signum з використанням умовних операторів:

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

4. Поліморфні типи

У мові Haskell використовується поліморфна система типів. По суті, це означає, що в мові присутні типові змінні. Розглянемо вже відому нам функцію tail, яка повертає перший елемент списку. Який тип цієї функції? Вона однаково застосовна і до списку цілих, і до списку символів, і до списку рядків:

```
Hugs> tail [1,2,3]
[2,3]
Hugs> tail [ 'a', 'b', 'з']
[ 'B', 'з']
Hugs> tail [ "list", "of", "lists"]
[ "Of", "lists"]
```

Функція tail має поліморфний тип: [a] -> [a]. Це означає, що вона приймає в якості аргументу будь-який список і повертає список того ж самого типу. Тут a позначає типову змінну, тобто мається на увазі, що замість неї можна підставити будь-який конкретний тип. Таким чином, запис [a] -> [a] задає ціле сімейство типів, представниками якого є, наприклад [Integer] -> [Integer], [Char] -> [Char], [[Char]] -> [[Char]] і т.п.

Аналогічно функція tail, що повертає перший елемент списку, має тип [a] -> a. Представниками цього сімейства є типи [Integer] -> Integer, [Char] -> Char і т.п.

Багато функцій, що працюють зі списками, парами і кортежами, мають поліморфні типи. Так, функція fst має тип (a, b) -> a (зауважте, що у визначенні цього типу використовуються дві типові змінні).

5. Призначені для користувача типи

Крім використання стандартних типів, програмісту надана можливість визначати свої власні, специфічні типи даних. Для цього використовується ключове слово data.

5.1 Пари

Для прикладу розглянемо визначення пари, дуже схожою на стандартну:

```
data Pair ab = Pair ab
```

Розглянемо цей код детально. Ключове слово data показує, що ми збираємося визначати тип даних. Потім слід назва цього типу, в даному випадку Pair (нагадаємо, що імена типів починаються з великої літери). Символи a й b, наступні за цим, є типовими змінними, які позначають

параметри типу. Таким чином, ми визначаємо структуру даних, параметрізованію двома типами a й b (нагадує шаблони класів мови C ++).

Після знака рівності ми вказуємо конструктори даних цього типу. В даному випадку у нас є єдиний конструктор Pair. (Ім'я конструктора даних не обов'язково має збігатися з ім'ям типу, але в нашому прикладі це виглядає природно.) Після імені конструктора даних ми знову пишемо a b, що означає, що для того, щоб сконструювати пару, нам потрібно два значення: одне, що належить типу a й інше, з типу b.

Це визначення вводить функцію *Pair :: a -> b -> Pair ab*, яка використовується для конструювання пар типу Pair. Завантаживши цей код в інтерпретатор, можна подивитися, як конструюються пари:

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

Функції, відповідні конструкторам даних, володіють тим властивістю, що їх можна використовувати при зіставленні зі зразком. Так, функції для отримання першого і другого елементу нашої пари можна визначити наступним чином:

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

Даний приклад, можливо, викликає питання: навіщо визначати свій тип Pair, якщо є стандартна можливість визначити пару? По-перше, з використанням типу Pair можна визначити набір функцій, що працюють тільки з цим типом і відокремити їх від функцій, які працюють з парами «взагалі». По-друге, зробивши один крок вперед, можна визначити деякі обмеження на одержувані пари, яких неможливо домогтися за допомогою стандартного типу. Наприклад, уявіть, що нам потрібен тип, який зберігає пару елементів одного і того ж типу. Його можна визначити наступним чином:

```
data SamePair a = SamePair aa
```

Тут тип має один параметр, однак конструктор даних приймає два параметра одного і того ж типу.

5.2 Множинні конструктори

У попередньому прикладі ми розглядали тип даних з єдиним конструктором. Також можливо (і часто дуже корисно) визначити тип з декількома конструкторами. Конструктори відокремлюються один від одного символом '|'.

Розглянемо тип Color, що представляє колір, з можливими значеннями Red, Green і Blue. Його можна визначити так:

```
data Color = Red | Green | Blue
```

Тут Color – назва типу, а Red, Green і Blue – конструктори даних. Зауважте, що цей тип не приймає параметрів. Такі типи називаються перелічуваних і відповідають конструкції enum в мові Сі. Такі типи дуже корисні. Наприклад, стандартний тип Bool визначено таким чином:

```
data Bool = True | False
```

Однак множинні конструктори можуть також приймати параметри. Так, можна помітити, що наш тип Color дозволяє визначити тільки три фіксованих кольору. Розширимо його так, щоб він дозволяв визначати довільний колір, що задається трьома цілими числами, відповідними рівнями червоного, зеленого і синього (стандартне rgb-подання):

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

Тут тип Color, крім стандартних квітів Red, Green і Blue (їх список можна, звичайно розширити), дозволяє визначати довільний колір за допомогою конструктора RGB, що приймає три цілих числа, що визначають rgb-компоненти кольору. Тоді, наприклад, функція для виділення red-компонента кольору запишеться так:

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

Типи з множинними конструкторами також можуть бути поліморфними. Розглянемо наступну проблему. Нехай функція повинна повернути деякий результат, або повідомити про помилку. Наприклад, функція для вирішення лінійного рівняння повертає знайдений корінь; функція для пошуку першого невід'ємного числа в списку повертає це число і т.п. Разом з тим рішення рівняння може не існувати, в списку не виявитися невід'ємних чисел і т.д. Як повідомити про це тому, хто викликав функцію? Іноді (як у випадку отримання невід'ємного елементу) можна ввести домовленість, що повернення будь-якого спеціального значення (наприклад, -1) означає «немає результату» (Багато функцій стандартної бібліотеки мови Сі так і надходять). Однак це не завжди можливо: в разі рішення лінійного рівняння такого виділеного значення не існує. Проблема витончено вирішується за допомогою стандартного типу Maybe, визначеного так: `data Maybe a = Nothing | Just a`

Тип Maybe (від англійського maybe - можливо) параметризрвані типовий змінної а й надає два конструктора: Nothing (англ. Нічого) для подання відсутності результату і Just (англ. Просто, в точності) для осмисленого результату. Тоді наші функції можна записати так:

```
- Функція повертає корінь рівняння ax + b = 0
solve :: Double -> Double -> Maybe Double
solve 0 b - Nothing
solve ab = Just (-b / a)

- Функція повертає перший ненегативний елемент списку
findPositive :: [Integer] -> Maybe Integer
findPositive [] = Nothing
findPositive (x: xs) | x> 0 = Just x
| otherwise = findPositive xs
```

Використання типу Maybe має низку переваг. Його використання явно показує, що функція може повернути «відсутність результату». У разі виділеного значення для того, щоб дізнатися, як функція повідомляє про це, необхідно вивчити документацію до функції (яка може бути відсутнім або бути невірною). З Maybe ця інформація міститься в типі функції і її може надати сам інтерпретатор. Більш того, при обробці повертається функції необхідно явно зробити зіставлення зі зразком, і якщо ми забудемо обробити випадок з Nothing, компілятор може видати попередження.

5.3 Класи типів

Класи типів будуть детально вивчатися пізніше. Тут ми дамо тільки базове уявлення про них, оскільки вони суттєво полегшують роботи з призначеними для користувача типами. Клас типів являє собою деякий безліч типів, що володіють рядом загальних властивостей. Наприклад, в клас типів Eq входять ті типи, для об'єктів яких визначено ставлення рівності, тобто, якщо змінні x і y належать одному і тому ж типу, що входить в клас Eq, ми можемо обчислювати вирази x = y і x / = y. Всі прості типи, а також списки і кортежі входять в цей клас, проте, наприклад, для функції відношення рівності не визначене і типи функцій не належать класу Eq.

Іншим важливим для нас класом є клас Show. У нього входять ті типи, об'єкти яких можуть бути перетворені в рядок для того, щоб її можна було відобразити на екрані. Прості типи, кортежі і списки входять в цей клас, тому інтерпретатор може надрукувати, наприклад, рядок. Функції не входять в цей клас.

За замовчуванням для користувача типи не входять ні до якого класу, тому значення цих типів не можна порівнювати і інтерпретатор не може надрукувати їх. Це, зрозуміло, незручно. Тому можна при визначенні типів задати їх приналежність бажаним класам. Для цього після визначення типу необхідно додати ключове слово deriving і в дужках перерахувати класи, до яких повинен належати тип.

Приклад:

```
- Тип, представляє час дня
data DayTime = Morning
| Afternoon
| Evening
| Might deriving (Eq, Show)
```

При визначенні типів в завданнях відносите їх до класів Eq і Show. Це істотно полегшить вашу роботу.

6. Функції вищого порядку

Розглянемо два завдання. Нехай заданий список чисел. Необхідно написати дві функції, перша з яких повертає список квадратних коренів цих чисел, а друга - список їх логарифмів. Ці функції можна визначити так:

```
sqrtList [] = []
sqrtList (x: xs) = sqrt x: sqrtList xs
logList [] = []
logList (x: xs) = log x: logList xs
```

Можна помітити, що ці функції використовують один і той же підхід, і вся різниця між ними полягає в тому, що в одній з них для обчислення елемента нового списку використовується функція квадратного кореня, а в іншій - логарифм. Чи можна абстрагуватися від конкретної функції перетворення елемента? Виявляється, можна. Згадаймо, що в Haskell функції є елементами «першого класу»: їх можна передавати в інші функції в якості параметрів. Визначимо функцію transformList, яка приймає два параметри: функцію перетворення і перетворений список.

```
transformList f [] = []
transformList f (x: xs) = fx: transformList f xs
```

Тепер функції sqrtList і logList можна визначити так:

```
sqrtList l = transformList sqrt l logList l = transformList log l
```

Або, з урахуванням каррінг:

```
sqrtList = transformList sqrt logList = transformList log
```

6.1 функція map

Насправді функція, повністю аналогічна transformList, вже визначена в стандартній бібліотеці мови і називається map (від англ. Map – відображення). Вона має наступний тип:

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

Це означає, що її першим аргументом є функція типу a -> b, що відображає значення довільного типу a в значення типу b (взагалі кажучи, ці типи можуть збігатися). Другим аргументом функції є список значень типу a. Тоді результатом функції буде список значень типу b.

Функції, подібні map, які беруть в якості аргументів інші функції, називаються функціями вищого порядку. Їх дуже широко використовують при написанні функціональних програм. З їх допомогою можна явно відокремити приватні деталі реалізації алгоритму (наприклад, конкретну функцію перетворення в map) від його високорівневою структури (поелементне перетворення списку). Алгоритми, представлені з використанням функцій вищого порядку, як правило, більш компактні і наочні, ніж реалізації, орієнтовані на конкретні зокрема.

6.2 функція filter

Наступним прикладом широко використовуваної функції вищого порядку є функція filter. По заданому предикату (функції, що повертає булевское значення) і списку вона повертає список тих елементів, які задовольняють заданому предикату:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x: xs) | px = x: filter xs
| otherwise = filter xs
```

Наприклад, функція, яка отримує зі списку чисел його позитивні елементи, визначається так:

```
getPositive = filter isPositive isPositive x = x> 0
```

6.3 Функції foldr і foldl

Більш складним прикладом є функції foldr і foldl. Розглянемо функції, які повертають суму і твір елементів списку:

```
sumList [] = 0
sumList (x: xs) = x + sumList xs
multList [] = 1
multList (x: xs) = x * multList xs
```

Тут також можна побачити загальні елементи: початкове значення (0 для підсумовування, 1 для множення) і функція, комбінує значення між собою. Функція foldr є очевидним узагальненням такої схеми:

```
Foldr :: (a -> b -> b) -> b -> [a] -> b
foldr fz [] = z
foldr fz (x: xs) = fx (foldr fz xs)
```

Функція foldr приймає в якості першого аргументу комбінує функцію

(зауважимо, що вона може приймати аргументи різних типів, але тип результату повинен збігатися з типом другого аргументу). Другим аргументом функції foldr є початкове значення для комбінування. Третім аргументом передається список. Функція здійснює «згортку» списку у відповідність до переданих параметрів.

Для того, щоб краще зрозуміти, як працює функція foldr, запишемо її визначення з використанням інфіксной нотації:

```
foldr fz [] = z
foldr fz (x: xs) = x 'f' (foldr fz xs)
```

Уявімо список елементів [a, b, c, ..., z] з використанням оператора:. Правило застосування функції foldr таке: всі оператори: замінюються на застосування функції f в інфіксне вигляді ('f'), а символ порожнього списку [] замінюється на початкове значення комбінування. Кроки перетворення можна зобразити так (припускаємо, що початкове значення дорівнює init)

```
[A, b, З, ..., z]
a: b: з: ...: []
a: (b : (C: (... (z: []) ...)))
a 'f' (b 'f '(c' f '(... (z' f 'init) ...)))
```

За допомогою функції foldr функції підсумовування і множення елементів списку визначається так:

```
sumList = foldr (+) 0
multList = foldr (*) 1
```

Розглянемо, як обчислюються значення цих функцій на прикладі списку [1, 2, 3]:

```
[1,2,3]
1: 2: 3: []
1: (2: (3: []))
1 + (2 + (3 + 0))
```

Аналогічно для множення:

```
[1,2,3]
1: 2: 3: []
1: (2: (3: []))
1 * (2 * (3 * 0))
```

Назва функції походить від англійського слова fold - згинати, складати (наприклад, аркуш паперу). Буква r в назві функції походить від слова right (правий) і показує асоціативність застосовуваної для згортки функції. Так, з наведених прикладів видно, що застосування функції групується вправо. Визначення функції foldl, де l вказує на те, що застосування операції

групується вліво, наведено нижче:

```
Foldl :: (a -> b -> a) -> a -> [b] -> a
foldl fz [] = z
foldl fz (x: xs) = foldl f (fzx) xs
```

Для асоціативних операцій, таких як додавання і множення, функції foldr і foldl еквівалентні, проте якщо операція не асоціативна, їх результат буде відрізнятися:

```
Main> foldr (-) 0 [1,2,3]
2
Main> foldl (-) 0 [1,2,3]
-6
```

Дійсно, в першому випадку обчислюється величина 1 - (2 - (3 -0)) = 2, а в другому - величина ((0 - 1) - 2) - 3 = -6.

6.4 Інші функції вищого порядку

У стандартній бібліотеці визначена функція zip. Вона перетворює два списки в список пар:

```
zip :: [a] -> [b] -> [(a, b)]
zip (a: as) (b: bs) = (a, b): zip as bs
zip _ _ = []
```

Приклад застосування:

```
Hugs> zip [1,2,3] [ 'a', 'b', 'c']
[(1, 'a'), (2, 'b'), (3, 'c')]
Hugs> zip [1,2,3] [ 'a', 'b', 'c', 'd'] [(1, 'a'), (2, 'b'), (3,
'c' )]
```

Зауважте, що довжина результуючого списку дорівнює довжині найкоротшого вихідного списку.

Узагальненням цієї функції є функція вищого порядку zipWith, «з'єднує» два списки за допомогою зазначеної функції:

```
zipWith :: (a-> b-> c) -> [a] -> [b] -> [c]
zipWith z (a: as) (b: bs) = zab: zipWith z as bs
zipWith _ = []
```

За допомогою цієї функції легко визначити, наприклад, функцію поелементного підсумовування двох списків:

```
sumList xs ys = zipWith (+) xs ys
```

або, з урахуванням каррінгу: `sumList = zipWith (+)`

7. Лямбда-абстракції

При використанні функцій вищого порядку часто необхідно визначати багато невеликих функцій. Наприклад, при визначенні функції getPositive нам довелося визначати додаткову функцію isPositive, яка потрібна тільки для того, щоб перевірити аргумент на позитивність. Зі збільшенням обсягу програми необхідність придумувати імена для допоміжних функцій все більше заважає. Однак в мові Haskell, як і в лежачому в його основі лямбдаобчисленні, можна визначати безіменні функції за допомогою конструкції лямбда-абстракції.

Наприклад, безіменні функції, що зводять свій аргумент в квадрат, додають одиницю і множать на два, записується таким чином:

```
\ X -> x * x
\ X -> x + 1
\ X -> 2 * x
```

Їх тепер можна використовувати в аргументах функцій вищих порядків.

Наприклад, функцію для зведення елементів списку в квадрат можна записати так:

```
squareList l = map (\ x -> x * x) l
```

Функція get Positive може бути визначена наступним чином:

```
getPositive = filter (\ x -> x> 0)
```

Можна визначати лямбдаабстракції для декількох змінних:

```
\ X y -> 2 * x + y
```

Лямбда-абстракції можна використовувати нарівні зі звичайними функціями, наприклад, застосовувати до аргументів:

```
Main> (\ x -> x + 1) 2
3
Main> (\ x -> x * x) 5
25
Main> (\ x -> 2 * x + y) 12
4
```

За допомогою лямбда абстракцій можна визначати функції. Наприклад, запис

```
square = \ x -> x * x повністю еквівалентна square x = x * x
```

8. Секції

Функції можна застосовувати частково, тобто не ставити значення всіх аргументів. Наприклад, якщо функція add визначена як

```
add x y = x + y
```

то можна визначити функцію inc, яка збільшить свій аргумент на 1 наступним чином:

```
inc = add 1
```

Виявляється, бінарні оператори, як вбудовані в мову, так і певні користувачам, також можна застосовувати лише до частини своїх аргументів (оскільки кількість аргументів у бінарних операторів дорівнює двом, ця частина складається з одного аргументу). Бінарна операція, застосована до одного аргументу, називається секцією.

Приклад:

```
(X +) = \ y -> x + y
(+ Y) = \ x -> x + y
(+) = \ X y -> x + y
```

Дужки тут обов'язкові. Таким чином, функції add і inc можна визначити так:

```
add = (+)
inc = (+1)
```

Секції особливо корисні при використанні їх в якості аргументів функцій вищого порядку. Згадаймо визначення функції для отримання позитивних елементів списку:

```
getPositive = filter (\ x -> x> 0)
```

З використанням секцій вона записується більш компактно:

```
getPositive = filter (> 0)
```

Функція для подвоєння елементів списку:

```
doubleList = map (* 2)
```

ДОДАТОК Г
ТЕОРЕТИЧНИЙ МАТЕРІАЛ
ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 4

Рекурсивні типи даних в мові Haskell.
Операції введення-виведення в мові Haskell

1. Визначення операторів

Бінарні оператори, такі як +, - і т. П. В мові Haskell є такими ж функціями, як і всі інші, за тим винятком, що для їх виклику можна використовувати інфіксне нотацію. Якщо взяти бінарний оператор в дужки, то для його виклику можна використовувати префіксних нотацію і звертатися з ним, як зі звичайною функцією. Так, такі пари записів еквівалентні:

```
2 + 2
(+) 2 + 2
x <y
(<) X y
x / = y
(/ =) Xy
```

Навпаки, будь-яку функцію, приймаючу два аргументи, можна використовувати в інфіксне стилі. Для цього її ім'я потрібно оточити зворотними лапками (символ '). Наприклад, якщо визначити функцію:

```
func xy = (x + y) / (xy)
```

то її можна викликати в наступних видах:

```
func 5 2
5 'func' 2
```

Далі, якщо в імені функції зустрічаються тільки «символи» (не букви і не цифри), то вона автоматично вважається інфіксне оператором. При визначенні її ім'я потрібно укладати в дужки. Наприклад, визначимо оператор «приблизно дорівнює», перевіряючий, що числа відрізняються не більше, ніж на 0.001: `(~ =) Xy = abs (xy) <0.001`

Тепер цей оператор можна використовувати так само, як і всі інші:

```
testApproxEqual xy = if x ~ = y then "equal"
else "not equal"
```

2. Рекурсивні типи

При визначенні типів даних в правій частині визначення можна використовувати визначається цією конструкцією тип. Це дає можливість визначати рекурсивні структури даних. Однією з основних таких структур є

дерево.

Визначимо бінарне дерево, в листі якого знаходяться елементи типу a, в такий спосіб:

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

Це визначення говорить, що дерево (Tree) є або листом (Leaf), т. Е. Вузлом, у якого немає нащадків, або гілкою (Branch), т. Е. Вузлом, у якого є ліве і праве піддерево. Зауважте, що в наведеному визначенні Leaf і Branch – конструктори даних, а Tree a, що зустрічається і в лівій, і в правій частині визначення – назва типу.

Робота з рекурсивними типами практично не відрізняється від роботи зі звичайними типами, за тим винятком, що практично всі функції, що працюють з рекурсивними типами, самі також рекурсивний.

Наприклад, визначимо функцію treeSize, що повертає кількість листя в дереві. Вона записується в такий спосіб:

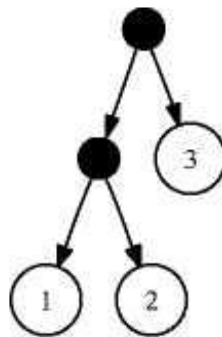
```
treeSize (Leaf _) = 1
treeSize (Branch lr) = treeSize l + treeSize r
```

Застосування цієї функції виглядає наступним чином:

```
Main> treeSize (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)) 3
```

Тут ми застосували її для дерева такого вигляду:

Іншим прикладом функції для роботи з деревами служить функція для отримання списку всіх листів дерева:




```
leafList (Leaf x) = [x]
leafList (Branch left right) = leafList left ++ leafList right
```

3. Списки як рекурсивні типи

Список також є рекурсивним типом. Розглянемо наступний поліморфний тип:

```
data List a = Nil | Cons a (List a)
```

Значення типу List a або порожньо (Nil), або містить елемент типу a і значення типу List a. Неважко помітити пряму аналогію зі списками, які також або порожні ([]), або містять голову типу a і хвіст, який є також списком. Подібність стане ще більш очевидною, якщо конструктор Cons записати в інфіксне вигляді: `data List a = Nil | a'Cons ' (List a)`

Таким чином, обліковий тип міг би бути визначений таким чином:

```
- Це не справжній код мови Haskell
data [a] = [] | a: [a]
```

Для значень типу List можна визначити всі функції, визначені для списків. Наведемо приклади функцій head, tail і map:

```
headList (Cons x _) = x
headList Nil = error "headList: empty list"
tailList (Cons _ y) = y
tailList Nil = error "tailList: empty list"
```

Проілюструємо роботу цих функцій:

```
Main> headList (Cons 1 (Cons 2 Nil))
1
Main> tailList (Cons 1 (Cons 2 Nil))
Cons 2 Nil
```

4. Синтаксичні дерева

Структури, подібні деревної, широко використовуються в програмуванні. Наприклад, результатом граматичного розбору програми в будь-якому компіляторі є синтаксичне дерево. Наведемо приклад такого дерева для виразів, що містять константи, символи додавання і множення:

```
data Expr = Const Integer
| Add Expr Expr
| Mult Expr Expr
```

З цього визначення видно, що вираз (Expression) є або целочисленной константою (Constant), або сумою або добутком двох виразів. Наприклад, для вираження 1 + 2 * (3 + 4) відповідне значення типу Expr має вигляд:

```
Add (Const 1) (Mult (Const 2) (Add (Const 3) (Const 4)))
```

Функцію обчислення значення виразу можна визначити наступним чином:

```
eval :: Expr -> Integer
eval (Const x) = x
eval (Add xy) = eval x + eval y
eval (Mult xy) = eval x * eval y
```

Можна розширити тип Expr, ввівши можливість використання змінних у виразах:

```
data Expr = Const Integer
| Var String
| Add Expr Expr
| Mult Expr Expr
```

Конструктор Var визначає змінну з вказаним ім'ям. Такий тип Expr дозволяє визначити, наприклад, функцію для диференціювання виразу:

```
diff :: Expr -> Expr
diff (Const _) = Const 0
diff (Var x) = Const 1
diff (Add xy) = Add (diff x) (diff y)
diff (Mult xy) = Add (Mult (diff x) y) (Mult x (diff y))
```

Перевіримо роботу цієї функції на прикладі диференціювання виразу x + x2 (не забудьте додати deriving (Show) після визначення типу Expr):

```
Main> diff (Add (Var "x") (Mult (Var "x") (Var "x"))) Add (Const
1) (Add (Mult (Const 1) (Var "x")) (Mult ( Var "x") (Const 1)))
```

Таким чином, в результаті диференціювання ми отримали вираз 1 + (1 · x + x · 1), яке є правильним, але, звичайно, потребує спрощення.

Іншим обмеженням функції diff є те, що вона не розрізняє, з якої змінної проводиться диференціювання. Відповідно, в реальності вона повинна приймати додатковий параметр - ім'я змінної диференціювання.

Завдання значень типу Expr безпосередньо досить незручно. В принципі, можна написати функцію, яка перетворює рядок виду "1 + x * y" в відповідне значення типу Expr. Проте написання такої функції досить трудомістким, тому студентам пропонується скористатися готовою функцією. Вона визначена в файлі expr. hs і називається parseExpr. У цьому ж файлі визначено тип Expr. Для того, щоб підключити цей файл, скопіюйте його в каталог, де знаходиться ваша програма і на її початку додайте рядок

```
import Expr
```

Функція parseExpr має наступний тип: `parseExpr :: String -> Expr`

За заданою рядку вона повертає її подання до вигляді значення типу Expr:

```
Main> parseExpr
Add (Const 1) (Var "x")
```

5. Модулі

Програми на мові Haskell складаються з набору модулів. Модулі служать двом цілям – управління просторами імен і створення абстрактних типів даних.

Модулі мають імена, що починаються з великої літери; в інтерпретаторі Hugs текст модуля повинен знаходитися в окремому файлі, ім'я якого збігається з ім'ям модуля. Цей файл повинен мати розширення * .hs.

З практичної точки зору модуль являє собою просто одне велике оголошення, що починається з ключового слова module. приведем приклад модуля з ім'ям Tree.

```
module Tree (Tree (Leaf, Branch), leafList) where
data Tree a = Leaf a | Branch (Tree a) (Tree a)
leafList (Leaf x) = [x]
leafList (Branch left right) = leafList left ++
leafList right
```

Модуль явно експортує Tree, Leaf, Branch і leafList. Що експортуються з модуля імена перераховуються в дужках після ключового слова module. Якщо це перерахування не вказано, за замовчуванням з модуля експортуються всі імена. Зауважте, що імена типу і його конструкторів повинні бути згруповані, як в конструкції Tree (Leaf, Branch). Як скорочення можна використовувати запис Tree (..). Також можливо експортувати тільки частина конструкторів даних.

Модуль Tree тепер можна імпортувати в будь-якій іншій модуль:

```
module Main where
import Tree (Tree (Leaf, Branch), leafList)
. . .
```

Тут ми явно вказали список імпортованих сутностей; якщо опустити його, імпортуються всі сутності, що експортуються з модуля.

Очевидно, якщо в двох імпортованих модулях містяться різні сутності з одним ім'ям, виникне проблема. Для того, щоб уникнути її в мові існує ключове слово qualified, за допомогою якого визначаються ті імпортовані модулі, імена об'єктів яких набувають вигляду: «Модуль.Об'єкт».

Наприклад, для модуля Tree:

```
module Main where import qualified Tree
leafList = Tree.leafList
```

6. Абстрактні типи даних

Використання модулів дозволяє визначати абстрактні типи даних,

тобто типи, внутрішня структура яких прихована від їх користувача. Наприклад, розглянемо найпростіший словник, по заданому слову повертає його значення:

```
module Dictionary where
data Dictionary = Dictionary [(String, String)]
getMeaning :: Dictionary -> String -> Maybe String
getMeaning (Dictionary []) _ = Nothing
getMeaning (Dictionary ((word, meaning): xs)) w
| w == word = Just meaning
| otherwise = getMeaning (Dictionary xs) w
```

Функція getMeaning по заданому словником і слову повертає знайдене значення (з використанням типу Maybe). Сам словник представляється списком пар.

Як створювати словник? Користувач цього модуля може визначити addWord, яка додає пару «словозначеніе» в словник і повертає модифікований словник:

```
import Dictionary
addWord (Dictionary dict) word meaning = Dictionary ((word, meaning): dict)
```

Тут користувач бачить уявлення словника у вигляді списку і може скористатися цим. Однак в подальшому ми можемо захотіти змінити уявлення словника. Список – досить неефективна структура даних для пошуку, якщо він стає великий. Набагато краще використовувати хештабліци або дерева пошуку. Однак, якщо уявлення типу Dictionary відкрито, ми не можемо змінити його без ризику порушити функціонування призначених для користувача програм.

Зробимо тип Dictionary абстрактним, щоб приховати від користувачів модуля його внутрішнє подання. Визначимо в модулі значення emptyDict, що представляє собою порожній словник і функцію addWord. Тоді користувачі зможуть спілкуватися з значеннями типу Dictionary тільки за допомогою дозволених функцій:

```
module Dictionary (Dictionary, getMeaning, addWord, emptyDict) where
data Dictionary = Dictionary [(String, String)]
getMeaning :: Dictionary -> String -> Maybe String
getMeaning (Dictionary []) _ = Nothing
getMeaning (Dictionary ((word, meaning): xs)) w
| w == word = Just meaning
| otherwise = getMeaning (Dictionary xs) w

addWord (Dictionary dict) word meaning = Dictionary ((word, meaning): dict)
emptyDict = Dictionary []
```

Абстрактні типи даних надають механізм приховування даних, який на мові об'єктно програмування називається инкапсуляцией.

7. Синоніми типів

У мові є можливість визначення синонімів типів, тобто Імен для часто використовуваних типів. Вони створюються за допомогою ключового слово type. Ось кілька прикладів:

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

Синоніми типів не визначають нових типів, а просто дають нові імена для вже існуючих типів. Наприклад, тип Person -> Name повністю еквівалентний типу (String, Maybe String) -> String. Однак їх використовують, оскільки, по-перше, вони допомагають дати більш короткі імена типам, а по-друге, підвищують рівень понимаемости коду.

8. Операції вводу-виводу

Система вводу-виводу в мові Haskell повністю функціональна, проте не поступається в можливостях системам вводавивода імперативних мов. В імперативних мовах програма являє собою послідовність дій, які зчитують і змінюють вміст навколишнього світу. Типовими діями є зчитування і установка глобальних змінних, запис в файл, зчитування з клавіатури і т. д. Такі дії також є і частиною Haskell, однак вони чітко відокремлені від чисто функціонального ядра мови.

Система вводу-виводу в мові Haskell побудована навколо концепції монад. Однак для програмування вводу-виводу розуміння монад потрібно не більше, ніж розуміння загальної алгебри для виконання простих арифметичних дій. Тому ми розглянемо систему вводу-виводу без прив'язки до монадам, які будуть вивчатися в лекціях.

За допомогою мови Haskell дії визначаються, а не виконуються. Визначення дії не означає, що воно виконується. Виконання дії відбувається за рамками обчислень виразів.

Дії є або атомарними, визначеними за допомогою системних примітивів, або послідовними композиціями інших дій. Монада вводавивода містить примітиви, які дозволяють створювати складові дії, аналогічно використанню ';' в імперативних мовах. Монада є «клеєм», що зв'язує дії в програмі.

8.1 Базові операції вводу-виводу

Кожна дія повертає значення. В системі типів це значення «позначено» типом IO, який відрізняє дії від інших значень. Наприклад, розглянемо функцію getChar:

```
getChar :: IO Char
```

IO Char показує, що getChar при виклику виконує деяку дію, яке повертає символ. Дії, які не повертають результату, використовують тип IO (). Символ () означає порожній тип (схожий на тип void в мові Сі). Наприклад, функція putChar:

```
putChar :: Char -> IO ()
```

Вона приймає символ і не повертає нічого цікавого.

Дії зв'язуються один з одним за допомогою оператора >> =. Однак ми будемо використовувати т. Н. doнотацію. Ключове слово do починає послідовність операторів, які виконуються по порядку. Оператор може бути або дією, яким зразком, пов'язують з результатом дії за допомогою <- . doнотація використовує ті ж правила вирівнювання, що і ключові слова let або where. Ось проста програма, яка зчитує символ і друкує його:

```
main :: IO () main = do c <-getChar putChar c
```

Використання імені main тут не випадково: функція main модуля Main є точкою входу в програму на мові Haskell, подібно функції main в Сі. Її тип повинен бути IO (). Представлена програма виконує дві дії послідовно: зчитує символ, пов'язує результат зі змінною c і потім друкує символ.

Як повернути значення з послідовності дій? Наприклад, нам необхідно визначити функцію ready, яка зчитує символ і повертає True, якщо він дорівнює 'y':

```
ready :: IO Bool
ready = do c <- getChar
c == 'y'- -Помилка!!!
```

Це не працює, оскільки другий оператор в do є просто булевих значенням, а не дією. Нам потрібно взяти булевское значення і створити дію, яке нічого не робить, але повертає це булевское значення в якості результату. Для цього служить функція return: `return :: a -> IO a`

Функція return завершує послідовність дій. Таким чином, ready визначається так:

```
ready :: IO Bool
ready = do c <- getChar
return (c == 'y')
```

Тепер можна визначити більш складні функції вводавивода. Функція getLine, що повертає рядок, зчитану з клавіатури з символом кінця рядка в якості завершального:

```
getLine :: IO String
getLine = do c <- getChar
         if c == '\ n'
 then return ""
 else do l <- getLine
  return (c: l)
```

Функція return вводить звичайне значення в царство дій введення-виведення. Як щодо зворотного напрямку? Чи можна виконати дію вводавивода в звичайному вираженні? Виявляється, немає! Функція, така як f :: Int -> Int не може виконувати операцій введення-виведення, оскільки IO не з'являється в типі її значення, що повертається.

8.2 Стандартні операції вводу-виводу

Розглянемо наступні дії і типи для роботи з файловим вводомвиводом (вони визначені в модулі IO):

```
type FilePath = String --імена файлів в файлову систему
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Щоб відкрити файл, використовується функція openFile, якій передається ім'я файлу і режим, в якому його необхідно відкрити. При цьому створюється дескриптор файлу (типу Handle), який потім необхідно закрити за допомогою функції hClose.

Для зчитування з файлу символу і рядки служать наступні функції:

```
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
```

Для запису в файл використовуються функції:

```
hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
```

Для зчитування з клавіатури і виведення на екран використовуються наступні функції:

```
getChar ::  IO Char
getLine ::  IO String
putChar ::  Char -> IO ()
putStr ::  String -> IO ()
```

Крім того, дуже корисна наступна функція:

```
hGetContents :: Handle -> IO String
```

Вона зчитує весь файл як одну велику рядок. На перший погляд ця функція дуже неефективна, однак в дійсності, изза використання

відкладених обчислень, з файлу вважається стільки символів, скільки необхідно, але не більше.

8.3 Приклад

Запишемо програму копіювання файлів. Вона зчитує з клавіатури імена двох файлів, вихідного і цільового, і копіює один файл в інший.

```
    --Функція друкує запрошення, зчитує ім'я файлу
    --І відкриває його в зазначеному режимі
    getAndOpenFile prompt mode = do putStr prompt
    name <-getLine openFile name mode
    main = do fromHandle <- getAndOpenFile "Copy from:" ReadMode
toHandle <- getAndOpenFile "Copy to" WriteMode contents <-
hGetContents fromHandle
    hPutStr toHandle contents
    hClose toHandle
    putStr "Done."
```

Незважаючи на те, що ми використовуємо функцію hGetContents, весь вміст файлу не буде знаходиться в пам'яті, оскільки воно буде прочитуватися в міру необхідності і записуватися на диск. Це дозволить копіювати навіть великі файли, обсяг яких перевищує обсяг оперативної пам'яті комп'ютера. Вихідний файл буде неявно закритий, коли з нього вважається останній символ.

Для доступу до параметрів командного рядка програми можна використовувати наступну функцію, певну в модулі System:

```
getArgs :: IO [String]
```

Ця функція повертає список рядків, що є параметрами командного рядка, подібно масиву argv в програмах на Сі. Програму копіювання тоді можна визначити так:

```
main = do args <- getArgs
copyFile putStr
"Done."
copyFile [from, to] = do fromHandle <-openFile from ReadMode
toHandle <-openFile to WriteMode
contents <-hGetContents fromHandle
hPutStr toHandle contents
hClose toHandle
copyFile _ = error "Usage: copy <from> <to>"
```

Ця програма приймає імена вихідного і цільового файлів з командного рядка. Функція copyFile друкує повідомлення про помилку, якщо в програму передано невірне кількість аргументів.

9. Створення виконуваних програм

До сих пір ми виконували програми на мові Haskell з використанням інтерпретатора. Однак існує можливість створювати окремі виконувані програми, для виконання яких не потрібне середовище інтерпретатора. Для цього використовується компілятор Glasgow Haskell Compiler, що викликається за допомогою команди ghc.

Для того, щоб скомпілювати набір модулів в виконувану програму, повинен бути визначений модуль з ім'ям Main, в якому необхідно визначити функцію main :: IO (). Цей модуль слід помістити в файл Main.hs. Для компіляції необхідно ввести в командному рядку наступну команду:

```
ghc make Main.hs
```

У разі, якщо програма містить помилки, інформація про них буде виведена на екран. Якщо помилок немає, компілятор створить виконуваний файл, який можна запускати на виконання.

Навчальне видання

Шевченко Ілона Володимирівна

Кузнецова Юлія Анатоліївна

Сьомочкін Максим Олександрович

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ
(Частина 1. Функціональне програмування)

Редактор О. Ф. Серьожкіна

Зв. план, 2021
Підписано до видання 30.03.2021
Ум. друк. арк. 5,4. Обл.-вид. арк. 6,125. Електронний ресурс

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17

Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30 03.2001